PUCRS

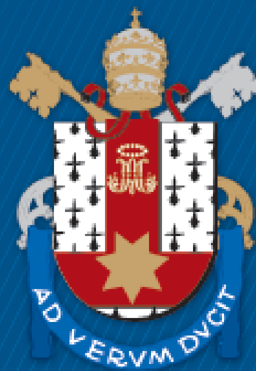
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

DINEI ANDRÉ ROCKENBACH

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR STREAM PARALLELISM ON GPUS

Porto Alegre
2021

PÓS-GRADUAÇÃO - *STRICTO SENSU*



Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR STREAM PARALLELISM ON GPUS

DINEI ANDRÉ ROCKENBACH

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Luiz Gustavo Leão Fernandes
Co-Advisor: Prof. Dalvan Jair Griebler

Porto Alegre
2020

Ficha Catalográfica

R682h Rockenbach, Dinei André

High-Level Programming Abstractions for Stream Parallelism on GPUs / Dinei André Rockenbach. – 2020.
161 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Co-orientador: Prof. Dr. Dalvan Jair Griebler.

1. Parallel programming. 2. parallel code generation. 3. GPU programming. 4. heterogeneous computing. 5. stream parallelism. I. Fernandes, Luiz Gustavo Leão. II. Griebler, Dalvan Jair. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Dinei André Rockenbach

High-Level Programming Abstractions for Stream Parallelism on GPUs

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on 27th November, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Marco Aldinucci (University of Torino)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS - Advisor)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS – Co-Advisor)

To my fiancée Cristiane, but also to everyone who participated, in any way, in this journey.

“The brick walls are there for a reason. The brick walls are not there to keep us out. The brick walls are there to give us a chance to show how badly we want something. Because the brick walls are there to stop the people who don’t want it badly enough. They’re there to stop the other people.”
(Randy Pausch, The Last Lecture)

ACKNOWLEDGMENTS

I would like to thank a number of people that contributed to the conclusion of this work. First, I would like to thank my advisors for the many hours spent guiding me towards an improved version of myself. Second, my peers and colleagues for the valuable feedback and for helping me in many occasions. Last, but not least, I would like to thank my family and my fiancée for all the support and for providing endless motivation to overcome the difficulties. This work was partially supported by funding from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and Hewlett Packard Brasil LTDA under Brazilian Informatics Law (Law nº 8.2.48 of 1991).

ABSTRAÇÕES DE PROGRAMAÇÃO DE ALTO NÍVEL PARA PARALELISMO DE FLUXO CONTÍNUO DE DADOS EM GPUS

RESUMO

O crescimento e disseminação das arquiteturas paralelas têm conduzido a busca por maior poder computacional com hardware massivamente paralelo tais como as unidades de processamento gráfico (GPUs). Essa nova arquitetura de computador heterogênea composta de unidade de processamento central (CPUs) com múltiplos núcleos e GPUs com muitos núcleos se tornou comum, possibilitando novas aplicações de software tais como carros com direção autônoma, *ray tracing* em tempo real, aprendizado profundo, e realidade virtual (VR), que são caracterizadas como aplicações de processamento de fluxo contínuo de dados. Porém, esse ambiente heterogêneo apresenta um desafio adicional para o desenvolvimento de software, que ainda está no processo de adaptação para o paradigma de processamento paralelo em sistemas com múltiplos núcleos, onde programadores têm a disposição várias interfaces de programação de aplicações (APIs) que oferecem diferentes níveis de abstração. A exploração de paralelismo em GPU é feito usando ambos CUDA e OpenCL pela academia e pela indústria, cujos desenvolvedores precisam lidar com conceitos de arquitetura de baixo nível para explorar o paralelismo de GPU eficientemente em suas aplicações. Existe uma carência de abstrações de programação paralela ao: 1) paralelizar código para GPUs, e 2) necessitar abstrações de programação de alto nível que lidam com o paralelismo de CPU e GPU combinados. Infelizmente, os desenvolvedores precisam ser programadores especialistas em sistemas operacionais e conhecer a arquitetura do hardware para permitir a exploração eficiente de paralelismo. Como contribuição à primeira carência, criou-se a GSPARLIB, uma nova biblioteca de programação paralela estruturada para explorar paralelismo de GPU que provê uma API de programação unificada e um ambiente de execução agnóstico ao *driver* da plataforma de hardware. Ela oferece os padrões paralelos Map e Reduce sobre os *drivers* CUDA e OpenCL. O seu desempenho

foi avaliado comparando com APIs do estado da arte, onde experimentos revelaram um desempenho comparável a eficiente. Como contribuição à segunda carência, estendeu-se a linguagem específica de domínio (DSL) SPar, que já foi testada e provada como sendo de alto nível e produtiva para expressar paralelismo de fluxo contínuo de dados com anotações C++ em CPUs de múltiplos núcleos. Neste trabalho, foram propostas e implementadas novas anotações que aumentam a expressividade para combinar o paralelismo de fluxo contínuo de dados em CPUs existente com o paralelismo de dados em GPUs. Também foram providenciadas novas regras de transformação baseadas em padrões, que foram implementadas no compilador almejando transformações automáticas de código-fonte para código-fonte usando a GSPARLIB para exploração de paralelismo de GPU. Os experimentos demonstram que o compilador da SPar é capaz de gerar padrões paralelos de paralelismo de fluxo contínuo de dados e de dados sem nenhuma redução de desempenho significativa quando comparada com código escrito pelo programador. Graças a esses avanços na SPar, este trabalho é o primeiro a prover anotações C++11 de alto nível como uma API que não requer refatoração significativa de código em programas sequenciais, para permitir a exploração de paralelismo em CPU de múltiplos núcleos e GPU de muitos núcleos em aplicações de processamento de fluxo contínuo de dados.

Palavras-Chave: Programação paralela, geração de código paralelo, programação para GPU, computação heterogênea, paralelismo de fluxo, aplicações de processamento de fluxo, linguagem específica de domínio, padrões de programação paralela, biblioteca de esqueletos algorítmicos, esqueletos algorítmicos, C++, Map, Reduce.

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR STREAM PARALLELISM ON GPUS

ABSTRACT

The growth and spread of parallel architectures have driven the pursuit of greater computing power with massively parallel hardware such as the Graphics Processing Units (GPUs). This new heterogeneous computer architecture composed of multi-core Central Processing Units (CPUs) and many-core GPUs became usual, enabling novel software applications such as self-driving cars, real-time ray tracing, deep learning, and Virtual Reality (VR), which are characterized as stream processing applications. However, this heterogeneous environment poses an additional challenge to software development, which is still in the process of adapting to the parallel processing paradigm on multi-core systems, where programmers are supported by several Application Programming Interfaces (APIs) that offer different abstraction levels. The parallelism exploitation in GPU is done using both CUDA and OpenCL for academia and industry, whose developers have to deal with low-level architecture concepts to efficiently exploit GPU parallelism in their applications. There is still a lack of parallel programming abstractions when: 1) parallelizing code on GPUs, and 2) needing higher-level programming abstractions that deal with both CPU and GPU parallelism. Unfortunately, developers still have to be expert programmers on system and architecture to enable efficient hardware parallelism exploitation in this architectural environment. To contribute to the first problem, we created GSParLib, a novel structured parallel programming library for exploiting GPU parallelism that provides a unified programming API and driver-agnostic runtime. It offers Map and Reduce parallel patterns on top of CUDA and OpenCL drivers. We evaluate its performance comparing with state-of-the-art APIs, where the experiments revealed a comparable and efficient performance. For contributing to the second problem, we extended the SPar Domain-Specific Language (DSL), which has been proved to be high-level and productive for expressing stream parallelism with C++ annotations in multi-core CPUs. In this

work, we propose and implement new annotations that increase expressiveness to combine the current stream parallelism on CPUs and data parallelism on GPUs. We also provide new pattern-based transformation rules that were implemented in the compiler targeting automatic source-to-source code transformations using GSPARLIB for GPU parallelism exploitation. Our experiments demonstrate that SPar compiler is able to generate stream and data parallel patterns without significant performance penalty compared to handwritten code. Thanks to these advances in SPar, our work is the first on providing high-level C++11 annotations as an API that does not require significant code refactoring in sequential programs while enabling multi-core CPU and many-core GPU parallelism exploitation for stream processing applications.

Keywords: Parallel programming, parallel code generation, GPU programming, heterogeneous computing, stream parallelism, stream processing applications, domain specific language, parallel patterns, skeleton library, algorithmic skeletons, C++, Map, Reduce.

LIST OF FIGURES

1.1 SPar's research framework. Adapted from [Gri16]. 21
2.1 Stream processing applications, from [AGT14]. 24
2.2 High-level representation of stream parallelism types. Adapted from [HSS+14]. 25
2.3 Overview of parallel patterns. Adapted from [Gri16] 26
2.4 Architectural differences between CPU e GPU. Extracted from [KmWH16]. . 29
2.5 Sequential pseudo-code annotated with SPar. Adapted from [GHDF18]. . . . 39
2.6 High-level representation of SPar compiler flow. 43
3.1 SkePU 2 compiler chain. Extracted from [ELK18]. 48
4.1 Methodology for defining GSParLib patterns. 57
4.2 Applying GSParLib in a sequential code. 58
4.3 Using FastFlow library to exploit stream parallelism in sequential source code. 59
4.4 Applying GSParLib in FastFlow code. 60
4.5 Overview of GSParLib APIs and parallel patterns. 62
4.6 UML class relationships of the Driver API. 66
4.7 UML class relationship of the Pattern API. 71
4.8 Default vs batched flow of execution. 75
4.9 Performance results of the vector sum algorithm using Map and Reduce. . . . 90
4.10 Performance results of the matrix multiplication algorithm using Map. 92
4.11 Performance results of the Mandelbrot set using Map. 94
5.1 Flow of the new transformation rules presented in Section 5.4. 111
5.2 Performance results of Mandelbrot Streaming for the small workload. 115
5.3 Performance results of Mandelbrot Streaming for the medium workload. . . . 116
5.4 Performance results of Mandelbrot Streaming for the large workload. 117
5.5 Latency profiling of SPar Mandelbrot Streaming with the medium workload. . 118
5.6 Latency profiling of different batch sizes in SPar (CUDA) Mandelbrot Streaming with the medium workload. 119
5.7 Effects of batching in SPar (CUDA) Mandelbrot Streaming for the medium workload. 121
5.8 Sample images of ray tracing applications. 122
5.9 Performance results of the ray tracing application for the small workload. . . . 125
5.10 Performance results of the ray tracing application for the medium workload. . 126
5.11 Performance results of the ray tracing application for the large workload. . . . 127

5.12 Lane detection steps output examples for frame 3 of KITTI 027 dataset. . . . 133
5.13 Performance results of lane detection for the Caltech dataset. 135
5.14 Performance results of lane detection for the KITTI dataset. 136
5.15 Performance difference of code generation in Mandelbrot Streaming. 138
5.16 Performance difference of code generation of the ray tracing application with the CUDA backend. 139
5.17 Performance difference of code generation in lane detection. 140
5.18 Applying SPar in sequential code. 141
5.19 Overview of SPar runtimes and supported parallel patterns. 144

LIST OF TABLES

2.1 Definitions for transformation rules from [GDTF17]. 41
3.1 Related Work of structured parallel programming APIs for GPGPU. 49
3.2 Related Work of annotation-based APIs for heterogeneous parallelism. 53
4.1 GSPARLIB Driver API analogous concepts. 64
4.2 Physical SLOC comparison between GPU programming APIs. 87
5.1 Definitions for transformation rules adapted from [GDTF17]. The definitions are applied in the order in which they are defined. 107
5.2 SEM in seconds for tests of Figure 5.2a. 115
5.3 SEM in seconds of tests for Figure 5.3a. 116
5.4 SEM in seconds of tests for Figure 5.4a. 117
5.5 SEM in seconds of tests for Figure 5.9a. 125
5.6 SEM in seconds of tests for Figure 5.10a. 126
5.7 SEM in seconds of tests for Figure 5.11a. 127
5.8 SEM in seconds of tests for Figure 5.13a. 134
5.9 SEM in seconds of tests for Figure 5.14a. 135
5.10 Physical SLOC comparison. 142

LIST OF ACRONYMS

ALU – Arithmetic Logic Unit

API – Application Programming Interface

AST – Abstract Syntax Tree

BLAS – Basic Linear Algebra Subprograms

CGI – Computer-Generated Imagery

CGMA – Compute to Global Memory Access

CINCLE – Compiler Infrastructure for New C/C++ Language Extensions

CPU – Central Processing Unit

CU – Compute Unit

CUDA – Compute Unified Device Architecture

DAS – Driving Automation System

DASP – Data Stream Processing

DNN – Deep Neural Network

DSL – Domain-Specific Language

DSP – Digital Signal Processor

FIFO – First In, First Out

FLOPS – Floating-point Operations per Second

FPGA – Field Programmable Gate Array

FPS – Frames Per Second

GB – GigaByte

GCC – GNU Compiler Collection

GCN – Graphics Core Next

GMAP – *Grupo de Modelagem de Aplicações Paralelas*

GMAVIS – Geospatial Map Visualization

GPGPU – General-Purpose computation on Graphics Processing Units

GPU – Graphics Processing Unit

GSPARLIB – GPU Stream Parallelism Library

HD – High Definition

HDARRAY – Heterogeneous Distributed Array

HPC – High Performance Computing

HSA – Heterogeneous System Architecture

HSAIL – Heterogeneous System Architecture Intermediate Layer

ILP – Integer Linear Program

INRIA – *Institut national de recherche en sciences et technologies du numérique* (National Institute for Research in Digital Science and Technology)

KIR – Kernel internal representation

LGPL – GNU Lesser General Public License

LIDAR – Light Detection and Ranging

MB – MegaByte

MIB – MebiByte

MIC – Many Integrated Core

MIT – Massachusetts Institute of Technology

ML – Machine Learning

MPI – Message Passing Interface

NN – Neural Network

NVRTC – NVIDIA Runtime Compilation

OOP – Object-Oriented Programming

OPENACC – Open Accelerators

OPENCL – Open Computing Language

OPENMP – Open Multi-Processing

OS – Operating System

PEPPHER – Performance Portability and Programmability for Heterogeneous Many-core Architectures

PNG – Portable Network Graphics

PPMSC – Parallel Programming Model Specific Code

PU – Processing Unit

REPARA – Reengineering and Enabling Performance And poweR of Applications

RGB – Red Green Blue

ROI – Region Of Interest

SAC – Single Assignment C

SAREK – Stream Architecture using Extensible Kernels

SAXPY – Single precision A X Plus Y

SEM – Standard Error of Mean

SGEMM – Single precision floating General Matrix Multiply

SIMD – Single Instruction, Multiple Data

SIMT – Single-Instruction, Multiple-Thread

SLO – Service Level Objective

SLOC – Source Lines of Code

SM – Streaming Multiprocessor

SP – Streaming Processor

SPAR – Stream Parallelism

SPOC – Stream Processing with OCaml

STL – Standard Template Library

TPU – Tensor Processing Unit

UML – Unified Modeling Language

VR – Virtual Reality

XMP – XscalableMP

CONTENTS

1 INTRODUCTION 18

2 BACKGROUND 23
2.1 STREAM PROCESSING APPLICATIONS 23
2.2 STRUCTURED PARALLEL PROGRAMMING 25
2.3 MULTI-CORE CPU AND MANY-CORE GPU 28
2.4 CONSOLIDATED GPU PARALLEL PROGRAMMING APIS 29
2.4.1 CUDA 30
2.4.2 OPENCL 32
2.4.3 OPENACC 34
2.4.4 STARPU 35
2.5 SPAR: HIGH-LEVEL PROGRAMMING ABSTRACTION FOR EXPRESSING STREAM PARALLELISM 38
2.5.1 SOURCE-TO-SOURCE TRANSFORMATION RULES 40
2.5.2 SPAR COMPILER 42

3 RELATED WORK 44
3.1 STRUCTURED PARALLEL PROGRAMMING APIS FOR GPU PROGRAMMING 45
3.2 ANNOTATION-BASED APPROACHES 50

4 GSPARLIB: UNIFIED GPU LIBRARY FOR STREAM PARALLELISM 54
4.1 MOTIVATION 54
4.2 PROGRAMMING WITH GSPARLIB 56
4.3 API AND RUNTIME DESIGN CHOICES 61
4.4 GSPARLIB: LOW-LEVEL DRIVER API 64
4.5 GSPARLIB: HIGH-LEVEL PATTERN API 70
4.5.1 MAP PATTERN 71
4.5.2 REDUCE PATTERN 75
4.5.3 PATTERN COMPOSITION 78
4.6 PROGRAMMABILITY CONSIDERATIONS 79
4.7 PERFORMANCE CONSIDERATIONS 88
4.8 FINAL REMARKS 95

5 HIGH-LEVEL STREAM AND DATA PARALLELISM FOR GPU WITH SPAR . . 97
5.1 EXTENDING SPAR LANGUAGE 97
5.2 SYNTAX OF THE NEW SPAR ATTRIBUTES 100
5.3 HOW TO ANNOTATE SEQUENTIAL CODES WITH SPAR 103
5.4 NEW COMPILER TRANSFORMATION RULES FOR SPAR 105
5.5 CODE GENERATION 110
5.6 SPAR PERFORMANCE CONSIDERATIONS 113
5.6.1 MANDELBROT STREAMING 114
5.6.2 RAY TRACING 121
5.6.3 LANE DETECTION 128
5.7 PERFORMANCE DIFFERENCE OF CODE GENERATION AND MANUAL IMPLEMENTATION 136
5.7.1 MANDELBROT STREAMING 137
5.7.2 RAY TRACING 139
5.7.3 LANE DETECTION 140
5.8 SPAR PROGRAMMABILITY CONSIDERATIONS 141
5.9 FINAL REMARKS 143

6 CONCLUSION 145
6.1 LIST OF PUBLISHED PAPERS 147

REFERENCES 149

1. INTRODUCTION

Since the dawn of the information age, the semiconductor industry has been guided by Moore's law [Moo65], crunching more transistors in a single chip. At the beginning of the XXI century, physical limitations have imposed restrictions on the possibility to increase the clock speed of these components. Then, the semiconductor industry started to integrate more and more processing cores in a single chip to increase the computing power [Pac11], creating the multi-core processors.

Until the XXI century, most software developers did not worry much about the applications' performance since faster machines appeared that were able to execute the same code faster [KmWH16]. Although parallel programming dates back to the 1960s, it became more important in the last years because this slow down in increasing processor's clock speed started in 2003 and stabilized in 2005, when the first multi-core CPU (Central Processing Unit) arrived in the desktop computers and became widespread [GM12]. Consequently, to speed up software performance, parallelism exploitation becomes mandatory.

The multi-core CPUs attempt to keep the sequential execution speed while also providing parallel processing. On the other hand, the many-core architecture, whose main player is the GPU (Graphics Processing Unit), focused on providing massive parallelism. This massively parallel hardware offers high-performance throughput concerning floating-point operations per second (FLOPS). In recent years, the maximum performance of GPUs in FLOPS has been around 10 times more than those of CPUs [KmWH16]. Since its cores are simpler and operate at lower clock speed, it is worth noting that any computing problem would not work better on the GPU than CPU, although GPUs have more FLOPS and thousands of GPU cores available. Differently, multi-core CPUs provide robust cores and larger cache memories to maintain the speed of running sequential programs.

This evolution in hardware technology enabled novel software applications to become reality, such as self-driving cars [HLLR14], real-time ray tracing [HAM19], deep learning, and virtual reality (VR). Many of these are stream processing applications, on which the data to be processed is received continuously from a data producer, such as cameras and sensors [TA10]. Each data item generated by the data producers is pushed into the input stream of a stream processing application. This application applies a sequence of operators over each data item and may produce a stream of output items, which can be stored or handed to another application. For example, self-driving cars contains many sensors and cameras, which generates data to be processed by a sequence of operations that identify lanes and objects, and ultimately control the car based on the analysis performed by these operators. The ray tracing technique is applied in a sequence of frames to generate a video or a video-game scene, which must take into account the inputs from the user that is playing the media. The virtual reality processes the data coming from device sensors to generate

realistic imagery. These kind of application usually have strict performance requirements such as latency or throughput. In turn, this requires programmers to efficiently utilize the underlying hardware.

The called multi-core machines or multi-core systems were the semiconductor industry's response to the desire of developers and users for faster machines when the clock rates stalled. However, this caused a big impact on software development [KmWH16]. Developers, which were used to sequential software development following von Neumann's architecture, have now to learn how to develop software specifically to take advantage of parallel processing. Currently, although the vast majority of programming languages already have native support for parallel processing, this is still a cumbersome task for inexperienced programmers.

More recently, the rise of massively parallel hardware and the performance differences of the multi-core and many-core architectures led developers to move computationally intensive (parallel) parts of the program to accelerators (such as GPUs). Programming applications to exploit more than one parallel architecture is known as heterogeneous parallel programming. In this sense, programming for many-core hardware poses additional challenges concerning parallel programming for multi-core machines, due to the differences in the architectural design.

The algorithmic skeletons [Col89, Col04] and parallel patterns [MSM04] are approaches to ease parallel software development. These algorithmic structures provide benefits such as simplified programming, increased portability and code re-use, improved performance, and semi-automatic optimizations [Col04]. These approaches are also known and referred to as structured parallel programming approaches [MRR12]. The parallel patterns are classified according to the type of parallelism they exploit. For example, stream processing applications expose stream parallelism, on which each processing step can process different data at the same time. On the other hand, applications that apply the same operation in different subsets of the same data expose data parallelism, which is suitable for processing in many-core architectures. In this context, software developers concerns are also separated. System programmers are those who develop tools or use low-level approaches for efficient parallelism exploitation, and application programmers are those who develop applications focusing in business rules and leverages the system programmers' tools to exploit the underlying hardware. Therefore, targeting both application and system programmers, the industry and academia provided several tools based on parallel patterns to assist in the task of developing parallel software [Ora20, Mic17, ADKT17, Rei07].

The *de facto* standard APIs for general purpose GPU programming (GPGPU) are CUDA and OpenCL. The first is a proprietary technology of NVIDIA and only supports devices from this manufacturer while the second is an open-source specification aiming at code portability among different manufacturers. Although CUDA offers an API with a higher-level of abstraction than OpenCL, it still requires programmers to refactor the application code and

be concerned about hardware details in order to exploit GPU parallelism. The programmer must worry about moving data between the main memory and the GPU memory, calculating the parallelism degree according to the device limits, invoking the kernel asynchronously in a special syntax, and releasing resources associated to the GPU device. If the programmer has to exploit both multi-core and many-core parallelism, more challenges arise since not all objects from the GPU programming APIs are thread-safe. Consequently, operating system synchronization mechanisms are necessary to synchronize the computation (e.g. by using mutexes).

Given that most of the structured parallel programming tools were focused in multi-core and cluster architectures, the computational power of GPUs sparked interest to support heterogeneous parallelism programming in systems composed of multi-core CPUs with many-core GPUs. While tools such as SkePU [ELK18] and SkelCL [SKG11] does not offer efficient abstractions for stream parallelism, other tools such as the compilers from StreamIt to CUDA [UGT09, HSW+11] does not consider the data parallelism exposed by the stream processing applications. Even tools that support stream and data parallelism, such as FastFlow [APD+15], requires significant code refactoring in order to exploit the heterogeneous hardware.

Ideally, the tools developed by system programmers should provide efficient abstractions that does not requires stream processing application programmers to learn hardware details in order to exploit the parallelism available in the computer architecture. To this end, SPar [GDTF17] is a domain-specific language (DSL) compatible with C++11 focused on expressing stream parallelism in a simple way. It offers high-level abstractions that allows programmers to exploit parallelism in multi-core processors by simply adding C++11 annotations to the sequential source code. It has demonstrated the capability of offering high productivity while maintaining a performance very close to other frameworks [GHDF18, GHDF17, GHL+17]. Therefore, the main question that drives this research is, **can we provide high-level abstractions like SPar offers for multi-core CPUs while exploiting combined stream and data parallelism in heterogeneous computer architectures composed of multi-core CPUs and many-core GPUs?**

For this research question, Figure 1.1 highlights the contributions of our work (in dashed black line) within the larger research framework proposed by [Gri16]. At this moment, SPar generates C++ code with FastFlow [GDTF17], TBB [HGDF20], and OpenMP [Hof20] libraries to exploit stream parallelism in multi-cores, and DSParLib [Pie20] for cluster (distributed memory) architectures. It is worth mentioning that the work from Löff [Lö20], which aims at exploiting data parallelism using SPar abstractions for multi-cores, was developed concurrently with this work. There are similarities between our works, but they were developed independently and are mostly unrelated. Yet for multi-core systems, SPar has been studied to cover autonomously and abstractly support the number of replicas management [VGF20] and extends for service level objectives via code annotations [GSV+18, GVS+19].

The SPar compiler was generated from CINCLE (Compiler Infrastructure for New C/C++ Language Extensions). In the application layer, there is GMaVis, an high-level description language for geospatial visualization [Led16]. In this layer, applications are also end-user software that use SPar to exploit stream parallelism. Our work extends SPar language to express data parallelism and includes support for GPU architectures by generating parallel code using our new library called GSPARLIB. Using it as runtime library was important in the process of providing high-level programming abstractions and avoiding lock-in vendor since both CUDA and OpenCL are widely and efficiently used for GPU parallelism exploitation. Most of the tools for data parallelism in GPUs focus in one or another [NVI19, Lut15, SKG11, ZM11]. From the structured parallel programming approaches, SkePU offers support for both drivers, however, it lacks thread-safety which is necessary to integrate with other tools (such as SPar, FastFlow or TBB). Moreover, there are opportunities to extend the support space in the research framework for other architectures such as Field Programmable Gate Array (FPGA) and Tensor Processing Unit (TPU).

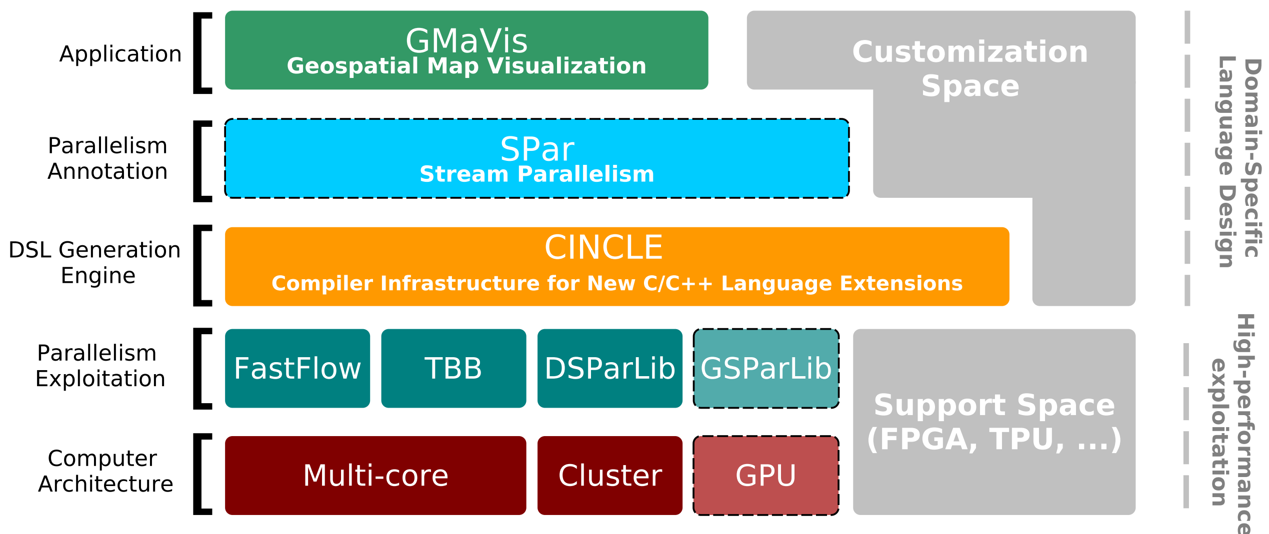


Figure 1.1: SPar's research framework. Adapted from [Gri16].

Our main goal is to design efficient and high-level parallel programming abstractions for expressing parallelism on stream processing applications targeting heterogeneous parallel computer architectures. Therefore, this work provides the following scientific contributions:

- a new structured parallel programming API (GSPARLIB) for heterogeneous parallel computer architectures equipped with multi-core CPU and many-core GPU, which supports different GPU drivers as well as provides an efficient and unified parallelism abstraction;
- an extension to SPar language to express data parallelism along with stream parallelism without substantial changes to the original syntax and semantics;

- a set of new definitions and source-to-source transformation rules targeting stream and data parallel patterns;
- a new compiler algorithm to automatic generate parallel code targeting data parallelism in GPU combined with stream parallelism on CPU; and
- a set of experiments using real-world applications to evaluate the performance and programmability of SPar and GSParLib.

The remainder of this document is organized as follows: Chapter 2 presents the background and context for this study, including an overview of solutions for GPU programming and the current SPar transformation rules and status. Chapter 3 presents other works related to ours, highlighting the differences. Chapter 4 introduces GSParLib, our structured programming library for GPU parallelism exploitation, including considerations and a performance comparison with other similar tools. Chapter 5 presents our extension to SPar language in order to support data parallel patterns in heterogeneous computer architectures by using GSParLib as runtime system. Finally, Chapter 6 presents the conclusions and future works.

2. BACKGROUND

Hardware improvements in the last years have been following the growing trend of data volume generated by the usage of digital technologies. A large number of tasks, formerly performed without extensive use of technology, is being computerized. However, the emergence of hardware with the potential to process this new volume of data brings in the challenge to develop software capable of exploiting the newly available computational power.

In this chapter we introduce the main concepts related to the objectives of our work. We start by introducing stream processing applications and their characteristics in Section 2.1, then we present the structured parallel programming perspective and the parallel patterns in Section 2.2. The concept of GPU parallel programming and the tools that ease this task are presented in Sections 2.3 and 2.4, respectively. Finally, Section 2.5 presents SPar, a DSL focused on stream parallelism, which we aim to improve in this study.

2.1 Stream processing applications

A stream is characterized by data made available in a continuous flow [TA10], usually generated by cameras, sensors, and other applications. These data streams must be collected, processed by a sequence of operators, and stored [VRJ+20]. In most cases, the business value of the streamed data can only be perceived with real-time processing and analysis, which leads to strict performance requirements for the applications that process these streams, called stream processing applications.

The stream processing applications are also the representatives of the computing paradigm known as Data Stream Processing (DaSP) [DM16, DM17]. These applications consume input data sources continuously and produce streams of output results [TAG+10]. This kind of applications are becoming increasingly common as the world gets more connected and more digital data are produced at an ever increasing pace [TAG+10]. There are examples of this kind of application in several domains, including data backup and compression, processing of data coming from monitoring sensors and logs, financial market, healthcare, and cryptography, among others. Many of these applications consume structured data, which share a common structure or schema, such as relational database-style records [AGT14]. By contrast, the most common commercial stream processing applications are those that processes unstructured data such as digital media in audio, image, and video formats, usually performing tasks such as compression, filter application, reproduction, etc. Figure 2.1 present some examples of stream processing applications. For example, the deep learning revolution in the last few years stands out precisely in media processing. This only reinforces the importance of tools for efficient programming of such applications.

Figure 2.1: Stream processing applications, from [AGT14].

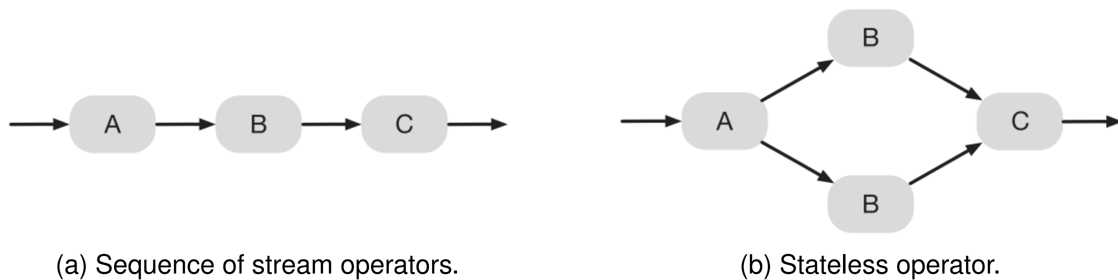
Differently from traditional applications, on which the volume of data to be processed is known or can be calculated, stream processing usually does not have a known or predefined end. In many cases this data flow comes from sensors measurements and there are strict requirements over the latency and throughput of the data processing, making it unfeasible to store the streamed data in a database and process them using traditional approaches [CJ09]. Moreover, the data input rate usually varies over the time, influenced by several different factors [TAG+10, CJ09, SRG+20]. An example of such application is a deep learning algorithm for object detection running over the images obtained by a camera attached to an autonomous system, such as a robot.

Hirzel [HSS+14] defines stream processing systems as runtime systems that execute stream graphs composed of operators (or stages) and FIFO (First In, First Out) communication queues. The stream input is an infinite sequence of data items or stream items, while the queues contain a finite number of items waiting to be consumed by each stream stage [SHGW15]. Each data items in the stream represents an atomic piece of data to be processed [AGT14].

Stream application programmers may trade throughput for latency by using batching in any of the stream operators. The batched operator does not start the computation until it has received enough items (according to the defined batch size). When the batch of items is full, the operator performs the computation of the entire batch at once. Thus, batching increases latency because each stream item waits in the batch queue until the batch is

filled up, and increases throughput by reducing warm-up, scheduling, and communication costs [HSS+14].

Each stream operator can process a different data item from the previous operator, therefore, the parallelism exploitation in stream processing is usually limited by the number of operators. Nonetheless, any stateless operator can be replicated to process multiple data items at the same time, further increasing the available parallelism degree. Figure 2.2 illustrates these two types of stream parallelism. The sequence of operators `A`, `B`, and `C` in Figure 2.2a can process different data items simultaneously. Figure 2.2b replicates the stateless operator `B`, thus increasing the throughput of this specific stage.



(a) Sequence of stream operators. (b) Stateless operator.

Figure 2.2: High-level representation of stream parallelism types. Adapted from [HSS+14].

2.2 Structured parallel programming

Programming parallel computers has been a challenging task since the dawn of parallel computing. However, the ubiquity of this kind of computers in the last two decades highlighted the importance of considering multiple computing cores while programming an application. The idea of providing algorithmic structures to ease the parallel programming burden is not new, but has been considered under different names [Gri16], such as algorithmic skeletons [Col89, Col04] and parallel patterns [MRR12].

Defining patterns of common programming tasks is an idea that comes from the design patterns largely used in software engineering. These patterns are mainly used for the development of software based on object-oriented programming (OOP) [GHJV94]. Using patterns for parallel programming aims to bring the benefits of the design patterns from the software engineering area to programmers targeting parallel computers. Some of these benefits are [Col04]: (a) simplified programming; (b) increased portability and re-use; (c) improved performance; and (d) more opportunities for automatic optimizations.

Figure 2.3 presents an overview with a visual representation of the main parallel programming patterns. The Pipeline pattern resembles much of a classic fordist assembly line, with well-defined tasks to be performed over data to produce transformed data, which are then passed over to the next stage [MRR12]. This pattern applies a sequence of operations

simultaneously to each data element [MSM04]. The parallelism exposed by this pattern is the possibility of computing each operation on a different data element at each given point in time. One example of Pipeline is applying a sequence of filters on the frames of a movie. Usually, the Pipeline pattern is suitable when all the stages present balanced workloads. If one stage becomes the bottleneck, the Farm pattern can be applied to increase the throughput of this given stage if it is stateless.

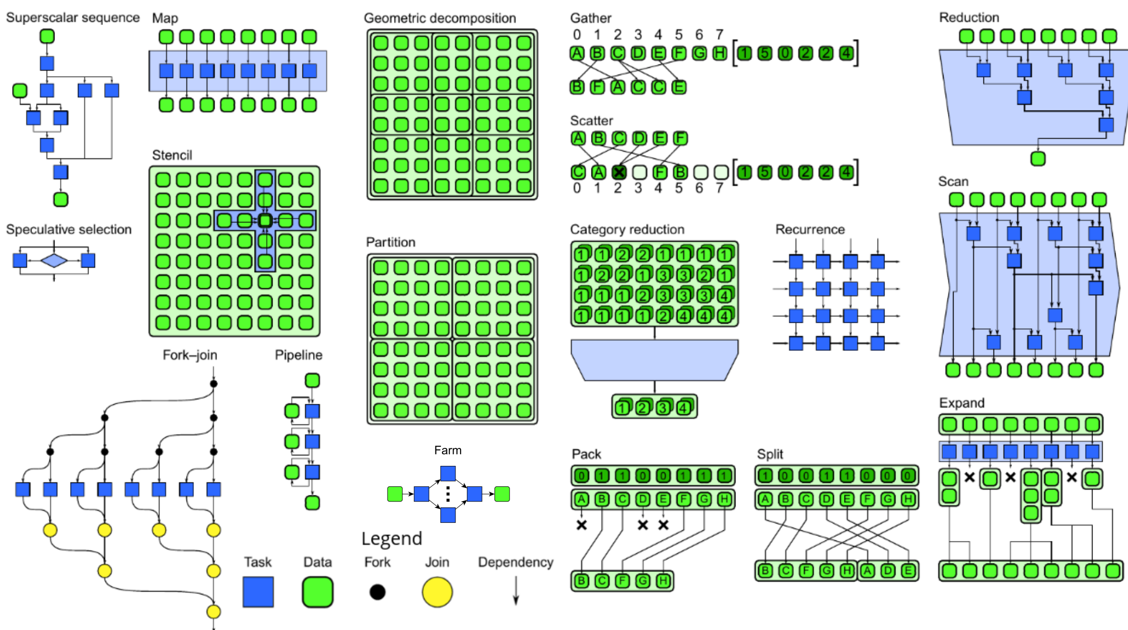


Figure 2.3: Overview of parallel patterns. Adapted from [Gri16]

The Farm pattern (also known as Split-Join [TKA02] or Fork/Join [MSM04]) is similar to a three-stage Pipeline, but it defines specialized stages such as: (a) the first stage is the emitter, which sends data to multiple parallel workers and plays such as the item scheduler; (b) the second stage defines the workers, which are replicated and process data simultaneously in *N* instances; and (c) the third stage is the collector, which collects and serializes the data from the parallel workers to provide extra features such as ordering and post-processing. The collector stage is optional. The Pipeline and Farm patterns are naturally suitable to stream processing [Gri16], since the data is streamed through the stages. These patterns may be directly associated with the stream parallelism types presented in Figure 2.2.

The Map pattern (also known as Loop Parallelism [MSM04]) is defined by a single task that can be performed in parallel over each data element on a set. It applies an operation over each input data element to transform it into the output data elements. In sequential programs, this kind of operation is usually performed by iterating over the data. Due to its characteristic that allows parallel computation with the absence of a rigid central coordination, this pattern is usually associated with the Single Instruction, Multiple Data (SIMD) or Single Program, Multiple Data (SPMD) models [MRR12, MSM04]. The problems on which the Map pattern can be applied are also known as embarrassingly parallel.

Due to its simplicity and scalability, the Map pattern is widely employed and provides the basis to other parallel patterns, such as the Stencil pattern. The Stencil pattern is effectively a Map with a different data access pattern. Even though the Stencil representation in Figure 2.3 may suggest a dependency among the computed data, the tasks executed are essentially independent, as long as the original data are available. In this pattern, the operation to be computed in parallel accesses a set of adjacent data. Applications may leverage data locality and cache memories to improve performance of applications that expose the Stencil pattern. There are many physics simulation algorithms (such as fluid flow and cellular automata) and image filters (such as gaussian blur and Sobel filter) that expose the Stencil pattern [MRR12].

Given that the Map pattern is obtained by parallelizing a loop with independent iterations, the Recurrence pattern is obtained by parallelizing a loop on which the iterations may depend of one another [MRR12]. However, these dependencies must follow a clear ordering based on a predefined interval. Recurrence also resembles the Stencil pattern, but unlike Stencil, where each operation only access the neighboring inputs, in the Recurrence pattern there may be dependencies over neighboring inputs and outputs. The Recurrence is used, for example, in algorithms for factoring matrices [GF11].

The Geometric Decomposition pattern is similar to Stencil in the sense that there is a separation of tasks based on the input data structure. In this pattern a geometric region of data is divided in subregions, facilitating the parallel computation of the function [GF11]. The special case of when the decomposed regions does not overlap receives the special name of Partition [MRR12]. This pattern, as well as Stencil, are specially suitable for algorithms such as Divide and Conquer [MSM04].

The Reduce pattern applies a function to combine the input data elements into a single output element [MRR12]. This pattern is only parallelizable if the combiner function is associative. Some parallel implementations require the combiner function to be commutative as well. A classic example of the Reduce algorithm is summing the elements of a vector, on which the sum operator combines the vector elements to obtain the final result. The Reduce pattern is commonly used in conjunction with Map, but it is also employed as a standalone pattern [Har07].

The Scan pattern can be seen as a special case of the sequential pattern Fold [MRR12]. The Fold pattern involves using a succession function to advance through stages, and the difference of Scan is that this function is associative, thus parallelizable. It worth noting that the parallel execution of Scan may require more work (executions of the succession function) than the sequential execution, limiting the scalability of this pattern and thus making it impossible to obtain the linear speedup. The visual representation of the Scan pattern in Figure 2.3 clearly resembles the Reduce pattern. However, unlike Reduce, in the Scan pattern each input produces an output. Moreover, in Scan there are dependencies of each

output element to the previous iterations, which can only be computed in parallel due to the associativity of the succession function.

The Pack pattern has the main objective of minimizing the algorithm memory consumption by removing unnecessary elements from the original collection [GF11]. By itself, Pack is not a good alternative to implement parallelism, but is particularly interesting when used together with other patterns, such as Map, to remove the unnecessary outputs from these patterns [MRR12].

Most applications cannot be expressed in terms of a single parallel pattern. Thus, multiple patterns are usually combined to express the parallelism on a single application. Applying a composition of patterns eases the programming effort and improves the parallelism exploitation.

The pragmatic manifesto of algorithmic structures by Cole [Col04] defines four distinct principles that should guide the design and development of tools based on structured parallel programming:

1. **Propagate the concept with minimal conceptual disruption**. The core pattern concepts must be kept unaltered, avoiding extra complexities in terms of conceptual baggage. The simplicity of the patterns should be maintained when conveying additional information related to the tool.

2. **Integrate ad-hoc parallelism**. The tool should permit interoperability with other tools used to express parallelism, instead of assuming that the structured parallel programming provides all the parallelism that is needed.

3. **Accommodate diversity**. The patterns should be implemented in a way that permits diverse ways of organization and versatile forms. The trade-off between abstract simplicity and pragmatic need for flexibility must be considered when developing tools for structured parallel programming.

4. **Show the pay-back**. The advantage of using a new tool should be clear to the user and must outweigh the overheads of such tool.

2.3 Multi-core CPU and many-core GPU

The inability to keep pace with the increase in speed of traditional processors not only driven the emergence of CPUs with multiple parallel processing cores, but has also given strength to the industry of “co-processors” or accelerators. The highly parallel nature of GPUs automatically qualified them as interesting accelerators to tasks with high parallelism. Figure 2.4 presents an overview of architectural differences between the CPUs and GPUs.

The modern CPUs architecture can be summarized in cache units (commonly L1, L2, and L3), control units, and ALUs (Arithmetic Logic Units). Beyond the oversimplification of Figure 2.4, the architecture of modern NVIDIA GPUs include streaming processors (SPs) or CUDA cores, analogous to ALUs, which contains an instruction cache and share control units [KmWH16]. SPs are grouped in streaming multiprocessors (SMs), which are analogous to the compute units (CUs) of AMD boards.

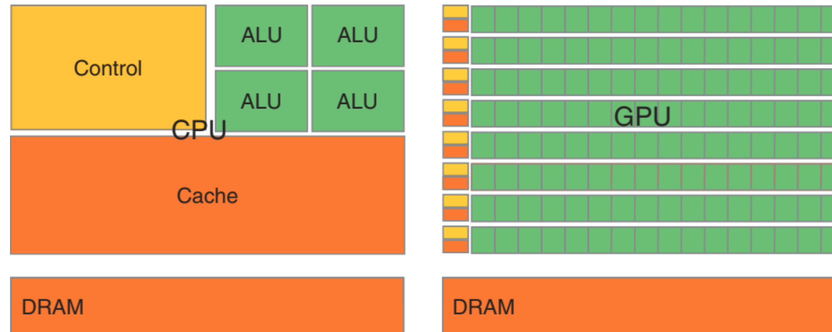


Figure 2.4: Architectural differences between CPU e GPU. Extracted from [KmWH16].

The operation of SMs uses the SIMT (Single-Instruction, Multiple-Thread) architecture, which can be described as SIMD (Single Instruction, Multiple Data) applied to the multithread context, on which the threads contain their own execution flow. In the SIMD architecture, threads that diverge in conditional ramifications are disabled (a technique known as masking) and wait until the ramification finishes execution [NVI18]. This is an important point to be considered to achieve a good performance, since the ideal performance is only achieved minimizing the number of disabled threads.

The SMs create groups of 32 threads called warps, which are scheduled to execution by the warp schedulers [NVI18]. Thread masking only occurs when the threads of the same warp diverge. Thus, the 32 threads of the same warp must agree in the execution flow to achieve a good performance. Contrary to CPUs, the GPUs does not make use of branch prediction and speculative execution [NVI18].

The architectural differences between CPUs and GPUs, summarized in Figure 2.4, inhibits the use of the GPU as exclusive processor and brings in programming for an heterogeneous system. In this environment, different software tasks are performed by different architectures [LD09], contrasting to homogeneous systems, which are composed of identical processors. Moreover, the parallelism exposed by CPUs, GPUs, and other accelerators poses an additional challenge to programming.

2.4 Consolidated GPU parallel programming APIs

The GeForce 3 GPU series, released by NVIDIA in 2001, enabled the use of GPUs through the OpenCL and DirectX APIs (Application Programming Interfaces), which were

specifically suited for graphics processing[1] [Har15]. The desire of developers back then to use the GPU processing to execute general purpose tasks led to the use of these APIs to copy numeric data into the GPU, and the hardware processes the calculation as if the numbers represented pixels colors, returning the results [LM01, RS01, HCSL02]. This way of exploiting the specialized hardware for general processing bring to life the concept of GPGPU (General-Purpose computation on Graphics Processing Units), i.e. using the GPU to perform tasks usually performed by the CPU [Har15].

Even though this approach of exploiting GPU worked initially, this trick required the representation of numeric data as if they were pixels colors, and this hard work did not seem attractive to many professionals. This landscape changed with the launch of the first GPU with support to DirectX 10 and the CUDA architecture (initially, Compute Unified Device Architecture, however this acronym was abandoned by NVIDIA in later versions) in 2007 [SK10], enabling GPU programming using a subset of the C language.

Efficiently programming heterogeneous parallel systems has proven to be a challenging task. Exploiting the GPU parallelism requires the use of yet another programming tool, in addition to those focused in multi-core parallelism. Moreover, the distinct memory architectures of CPU and GPU (as depicted in Figure 2.4) requires the programmer to explicitly copy data between the two platforms, which increases the program complexity. We will discuss some of the tools for abstracting these differences and combining multi-core and GPU parallelism in Chapter 3.

2.4.1 CUDA

With the introduction of the CUDA architecture in November 2006, exploiting the GPU parallelism was not restricted to professionals with large knowledge of the GPU architecture and the internals of their graphical APIs such as DirectX and OpenGL. Thus, it started to be widely used by C/C++ and Fortran programmers. The set of tools for CUDA programming involves a custom compiler (`nvcc`, profilers for performance analysis (NVIDIA Nsight Compute and NVIDIA Nsight Systems), performance counters as well as a bunch of libraries with different focuses. Because it is a proprietary technology of NVIDIA, only GPUs developed by the company offer support to CUDA.

CUDA offers two APIs: the higher-level Runtime API, which requires the use of the `nvcc` compiler and abstracts from the programmer tasks such as the driver initialzation, kernel compilation (which are performed in compilation time), and context management; and the lower-level Driver API, which is distributed as a static library that can be used in any modern C/C++ compiler (`-lcuda`), and offers an API for kernel compilation during runtime (`nvrtc`).

[1] https://web.archive.org/web/20031030015158/http://www.nvidia.com/object/geforce3_faq.html

The `nvcc` compiler distributed with the CUDA toolkit allows the developer to mark functions suitable for GPU execution using the declaration `__global__` (which are then called kernels). The kernels calls are performed using a special syntax which involves specifying the parallelism degree to be used in the execution, using the execution configuration syntax `<<<...>>>` between the kernel name and its parameters. When this call is performed, the threads are organized in thread blocks with up to three dimensions (for more details, see Section 2.3). Identifiers for each of these dimensions can be accessed using the special predefined variables `threadIdx`, `blockIdx`, and `blockDim`, inside the kernel [NVI18].

Listing 2.1 shows a sample vector sum application in CUDA using the Runtime API. In CUDA programming it is necessary to declare explicitly the code portions that will execute in the co-processor (by defining kernels). The `vecAdd` kernel is defined in line 2 of Listing 2.1. It receives a pair of vectors with `N` elements to be summed in parallel into the output vector. This simplified application assumes that `N` will be lower than the maximum size of each block (`maxThreadsPerBlock`) when calling the kernel in line 23. Other tasks that need to be explicitly done by the programmer are GPU memory allocation using the `cudaMalloc` function, copying memory between the main (host) memory and the device (GPU) memory using the `cudaMemcpy` function, and release the allocated memory using the `cudaFree` function.

```
1  #define N 10
2  __global__ void vecAdd(int *A, int *B, int *C) { // CUDA kernel definition
3    int tid = threadIdx.x; //handle the data at this index
4    if (tid < N)
5      C[tid] = A[tid] + B[tid];
6  }
7  int main() {
8    int host_a[N], host_b[N], host_c[N];
9    int *dev_a, *dev_b, *dev_c;
10   // fill the arrays 'a' and 'b' on the CPU
11   for (int i=0; i<N; i++) {
12     host_a[i] = -i;
13     host_b[i] = i * i;
14   }
15   // allocates memory on the GPU
16   cudaMalloc( (void**)&dev_a, N * sizeof(int) );
17   cudaMalloc( (void**)&dev_b, N * sizeof(int) );
18   cudaMalloc( (void**)&dev_c, N * sizeof(int) );
19   // copy the arrays 'a' and 'b' to the GPU
20   cudaMemcpy( dev_a, host_a, N * sizeof(int), cudaMemcpyHostToDevice );
21   cudaMemcpy( dev_b, host_b, N * sizeof(int), cudaMemcpyHostToDevice );
22   // calls the vecAdd kernel
23   vecAdd<<<1, N>>>( dev_a, dev_b, dev_c );
24   // copy the array 'c' back from the GPU to the CPU
25   cudaMemcpy( host_c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
26   for (int i=0; i<N; i++) // display the results
27     printf( "%d + %d = %d\n", host_a[i], host_b[i], host_c[i] );
28   cudaFree( dev_a ); // free the GPU memory
29   cudaFree( dev_b );
30   cudaFree( dev_c );
31   return 0;
32 }
```

Listing 2.1: Vector sum in CUDA Runtime API. Adapted from [SK10]

2.4.2 OpenCL

OpenCL (Open Computing Language) is an open industry standard framework that includes a language, API, libraries and a runtime system [Khr18, MGM+12]. It was initially proposed in 2008 by Apple (which still owns the trademark) and after received support by various companies, such as AMD (which oficcially recommends OpenCL as the way to explore parallelism in its CPUs and GPUs), IBM, Intel, and NVIDIA, among others. Nowadays the OpenCL specification is maintained by the Khronos Group.

The OpenCL objective is to provide code portability among different architectures and hardware [Khr18]. Differently from libraries which focuses in GPU parallelism, the idea behind OpenCL is that all available devices should be used to extract the maximum possible performance. In order to do so, an OpenCL application may follow the following steps [MGM+12]:

1. Discover the components of the heterogeneous systems and its characteristics.
2. Create instruction blocks (kernels) to run in each component.
3. Manage the data and components' memory to allow the computation execution.
4. Execute the blocks in the correct order and in the correct components.
5. Collect the results.

OpenCL defined that the compatible devices contains work-items (analogous to the threads of the CUDA architecture) organized in work-groups (analogous to the CUDA thread blocks). Just like the CUDA architecture, the work-groups and work-items are organized in a three-dimensional space, however, the unique identified of the thread can be obtained by calling the `get_global_id` function.

Listing 2.2 shows the OpenCL code for the same vector sum algorithm previously presented in CUDA language (Listing 2.1). The OpenCL code is considerably longer compared to the CUDA version. Most of these differences are due to the fact that the CUDA custom compiler (`nvcc`) allows the use of the Runtime API, which abstracts some boilerplate tasks from the programmer [KMSZ15].

```
1  const char *kernelSource = // OpenCL kernel definition
2  "__kernel void vecAdd(__global int *A, \n" \
3  " __global int *B, __global int *C, \n" \
4  " const unsigned int n) { \n" \
5  " int tid = get_global_id(0); \n" \
6  " if (tid < n) \n" \
7  "   C[tid] = A[tid] + B[tid]; \n" \
8  "}";
9  #include <CL/opencl.h>
10 int main() {
```

```
11   unsigned int N = 2048;
12   int host_a[N], host_b[N], host_c[N];
13   cl_mem dev_a, dev_b, dev_c;
14   // fill the arrays 'a' and 'b' on the CPU
15   for (int i=0; i<N; i++) {
16     host_a[i] = -i;
17     host_b[i] = i * i;
18   }
19   cl_int status;
20   cl_platform_id platform; // First platform
21   status = clGetPlatformIDs(1, &platform, NULL) ;
22   cl_device_id device; // First device
23   status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
24   // Create a context on the device
25   cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
26   cl_command_queue queue = clCreateCommandQueueWithProperties(context, device,
      0, &status);
27   size_t datasize = sizeof(int) * N;
28   dev_a = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
29   dev_b = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
30   dev_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
31   // Write data from host to device
32   status = clEnqueueWriteBuffer(queue, dev_a, CL_FALSE, 0, datasize, host_a, 0,
      NULL, NULL);
33   status = clEnqueueWriteBuffer(queue, dev_b, CL_FALSE, 0, datasize, host_b, 0,
      NULL, NULL);
34   // Create a program with provided source code
35   cl_program program = clCreateProgramWithSource(context, 1, (const
      char**)&kernelSource, NULL, &status);
36   status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
37   cl_kernel kernel = clCreateKernel(program, "vecAdd", &status);
38   status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_a);
39   status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_b);
40   status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_c);
41   status = clSetKernelArg(kernel, 3, sizeof(unsigned int), &N);
42   // Define an index space of work-items for execution
43   // A global size is not required, but can be used
44   size_t localSize[1], globalSize[1];
45   localSize[0] = 64; //Number of work-items in each local work group, must be
      divisible by globalSize
46   globalSize[0] = N; //Total number of work-items
47   // Execute the kernel
48   status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
      &localSize, 0, NULL, NULL);
49   // Read back the results
50   status = clEnqueueReadBuffer(queue, dev_c, CL_TRUE, 0, datasize, host_c, 0,
      NULL, NULL);
51   // Releases GPU resources
52   clReleaseKernel(kernel);
53   clReleaseProgram(program);
54   clReleaseCommandQueue(queue);
55   clReleaseMemObject(dev_a);
56   clReleaseMemObject(dev_b);
57   clReleaseMemObject(dev_c);
58   clReleaseContext(context);
59   return 0;
60 }
```

Listing 2.2: Vector sum in OpenCL. Adapted from [KMSZ15]

2.4.3 OpenACC

The OpenACC (from Open Accelerators), which was launched in 2011, is a programming model based in compiler directives to exploit the parallelism in parallel accelerator devices coupled to a main CPU host [Ope15a, Ope17]. It is considered a platform independent model, with a high-level abstraction level.

The OpenACC approach to express the parallelism in applications is based in 4 steps [Ope15a]: (a) improve the application performance as a whole, in order to identify the main regions which should be send to the accelerator device; (b) parallelize the identified loops using the OpenACC directives; (c) optimize the data location in memory in order to minimize unnecessary memory copies and try to guarantee that the data is available to the processor when necessary; (d) optimize the loops for executing in the accelerator, restructuring them to expose more parallelism or to reduce the data movements in memory.

The execution model of OpenACC is based in three levels of parallelism, which are defined as: `gang`, with less granularity; `worker`, for more granularity; and `vector`, which compute instructions in the SIMD model [Ope17]. Listing 2.3 shows an example of vector sum in OpenACC, which is considerably shorter than CUDA and OpenCL examples of this same application.

```
#define N 10
// OpenACC kernel definition
void vecAdd(int *A, int *B, int *C) {
  #pragma acc kernels loop independent copyin(A[0:N],B[0:N]), copyout(C[0:N])
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
int main() {
  int a[N], b[N], c[N];
  // fill the arrays 'a' and 'b' on the CPU
  for (int i=0; i<N; i++) {
    a[i] = -i;
    b[i] = i * i;
  }

  vecAdd( a, b, c );

  // display the results
  for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
  }
  return 0;
}
```

Listing 2.3: Vector sum in OpenACC

The OpenACC annotation in the C or C++ code are performed through the `#pragma acc` directive, followed by the directive name and an optional list of clauses. The clauses used in Listing 2.3 are: `kernels`, which activates the automatic identification of parallelism by the

compiler; `loop`, which informs the compiler that the next instruction is a loop; `independent`, which guarantees that the operations contained within the loop have no data dependency; and `copyin` and `copyout`, which defined the data that should be copied from the main memory to the accelerator memory and the data that should be copied back after the computation.

2.4.4 StarPU

StarPU is a runtime system that provides a unified execution model for heterogeneous hardware [ATNW11] developed by members of the National Institute for Research in Digital Science and Technology (Inria), France. Its development started in the PhD thesis of Cédric Augonnet [Aug11] using C language. The source code is open sourced under GNU Lesser General Public License (LGPL).

StarPU leverages a task-based execution flow to exploit parallelism in heterogeneous systems, with focus on task scheduling and memory management [ATAF14]. The scheduling algorithms can also be plugged in the StarPU open scheduling platform to customize the task scheduler [ATAF14]. It supports a large number of parallel architectures, such as multicore CPUs, GPUs (NVIDIA and OpenCL-compatible devices), Intel Xeon Phi, and Cell processors [ATAF14]. It runs on Linux, Mac OS, and Windows.

To make use of the GPU programming capabilities of StarPU, the kernel code for CUDA or OpenCL must be provided by the developer. In addition, for the programmer to use StarPU with CUDA, the code must be compiled with a CUDA compiler, such as `nvcc` [Sta20]. StarPU automatically handles memory transfers between the processing units (PUs) of the host processor and accelerators and schedules the tasks for execution. Listing 2.4 shows a sample matrix-vector multiplication adapted from the official repository of StarPU[2], which uses the OpenCL kernel depicted in Listing 2.5 to perform the computations.

```
1  #include <starpu.h>
2  #include <math.h>
3
4  struct starpu_opencl_program opencl_code;
5  void opencl_codelet(void *descr[], void *_args) {
6    (void)_args;
7    cl_kernel kernel;
8    cl_command_queue queue;
9    int id, devid, err, n;
10   cl_mem matrix = (cl_mem)STARPU_MATRIX_GET_DEV_HANDLE(descr[0]);
11   cl_mem vector = (cl_mem)STARPU_VECTOR_GET_DEV_HANDLE(descr[1]);
12   cl_mem mult = (cl_mem)STARPU_VECTOR_GET_DEV_HANDLE(descr[2]);
13   int nx = STARPU_MATRIX_GET_NX(descr[0]);
14   int ny = STARPU_MATRIX_GET_NY(descr[0]);
15   int ld = STARPU_MATRIX_GET_LD(descr[0]);
16
17   id = starpu_worker_get_id_check();
```

[2]https://scm.gforge.inria.fr/anonscm/git/starpu/starpu.git

```
  devid = starpu_worker_get_devid(id);

  err = starpu_opencl_load_kernel(&kernel, &queue, &opencl_code, "matVecMult",
   devid);
  if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

  n=0;
  err = clSetKernelArg(kernel, n++, sizeof(matrix), &matrix);
  err |= clSetKernelArg(kernel, n++, sizeof(vector), &vector);
  err |= clSetKernelArg(kernel, n++, sizeof(nx), (void*)&nx);
  err |= clSetKernelArg(kernel, n++, sizeof(ny), (void*)&ny);
  err |= clSetKernelArg(kernel, n++, sizeof(mult), &mult);
  err |= clSetKernelArg(kernel, n++, sizeof(ld), (void*)&ld);
  if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

  size_t global=nx*ny;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, NULL, 0, NULL,
   NULL);
  if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
  starpu_opencl_release_kernel(kernel);
}

static struct starpu_perfmodel starpu_matvecmult_model = {
  .type = STARPU_HISTORY_BASED,
  .symbol = "matvecmult"
};

static struct starpu_codelet cl = {
  .opencl_funcs[0] = opencl_codelet,
  .opencl_flags = {STARPU_OPENCL_ASYNC},
  .nbuffers = 3,
  .modes[0] = STARPU_R,
  .modes[1] = STARPU_R,
  .modes[2] = STARPU_RW,
  .model = &starpu_matvecmult_model
};

int main(void) {
  struct starpu_conf conf;
  starpu_conf_init(&conf);
  conf.ncpus = 0;
  conf.ncuda = 0;
  conf.nmic = 0;
  conf.nopencl = 1;

  int MX;
  float *matrix, *vector, *result;
  starpu_data_handle_t matrix_handle, vector_handle, result_handle;
  int ret, submit;

  MX = 20; // Matrix size (square)

  ret = starpu_init(&conf);
  if (STARPU_UNLIKELY(ret == -ENODEV)) {
    fprintf(stderr, "This application requires an OpenCL worker.\n");
    return 77; // Error code
  }

  matrix = (float*)malloc(MX * MX * sizeof(float));
```

```
  vector = (float*)malloc(MX * sizeof(float));
  result = (float*)malloc(MX * sizeof(float));

  // The code to fill the vector and matrix was supressed

  starpu_matrix_data_register(&matrix_handle, STARPU_MAIN_RAM,
   (uintptr_t)matrix, MX, MX, MX, sizeof(float));
  starpu_vector_data_register(&vector_handle, STARPU_MAIN_RAM,
   (uintptr_t)vector, MX, sizeof(float));
  starpu_vector_data_register(&result_handle, STARPU_MAIN_RAM,
   (uintptr_t)result, MX, sizeof(float));

  ret = starpu_opencl_load_opencl_from_file("matvecmult_kernel.cl",
   &opencl_code, NULL);
  STARPU_CHECK_RETURN_VALUE(ret, "starpu_opencl_load_opencl_from_file");

  struct starpu_task *task = starpu_task_create();
  task->cl = &cl;
  task->callback_func = NULL;
  task->handles[0] = matrix_handle;
  task->handles[1] = vector_handle;
  task->handles[2] = result_handle;

  submit = starpu_task_submit(task);
  if (STARPU_UNLIKELY(submit == -ENODEV)) {
    fprintf(stderr, "No worker may execute this task. This application requires
   an OpenCL worker.\n");
  } else {
    starpu_task_wait_for_all();
  }

  starpu_data_unregister(matrix_handle);
  starpu_data_unregister(vector_handle);
  starpu_data_unregister(result_handle);

  if (STARPU_LIKELY(submit != -ENODEV)) {
    FPRINTF(stdout, "Test finished");
  }

  free(matrix);
  free(vector);
  free(result);
  starpu_shutdown();

  return (submit == -ENODEV) ? 77 : 0;
}
```

Listing 2.4: Matrix vector multiplication using StarPU and OpenCL. Adapted from the official repository.

```
__kernel void matVecMult(const __global float *A, const __global float *X, int
    n, int m, __global float *Y, int ld) {
  const int i = get_global_id(0);
  if (i < m) {
    float val = 0;
    int j;
    for (j = 0; j < n; j++)
      val += A[i*ld+j] * X[j];
    Y[i] = val;
```

```
9      }
10 }
```

Listing 2.5: Matrix vector multiplication OpenCL kernel. Adapted from the StarPU repository.

The StarPU API provides some helper functions to common GPU programming tasks which abstracts some boilerplate code, such as `starpu_opencl_load_kernel`, which compiles the OpenCL kernel, creates a command queue, and loads the kernel object. Similarly, the memory objects (`cl_mem`) are wrapped in StarPU's data handles, which allows StarPU to perform automatic data copies between the host and device memories. However, some tasks involve direct calls to the OpenCL API, such as setting the kernel arguments and launching the kernel on the GPU. Therefore, StarPU provides good abstractions to task parallelism and scheduling but lacks high-level abstractions to the domain of GPU programming.

2.5 SPar: high-level programming abstraction for expressing stream parallelism

SPar [GDTF17] is a domain-specific language (DSL) focused on expressing stream parallelism and was created by Dalvan Griebler in his PhD thesis [Gri16]. The main drivers behind SPar are: (a) optimize programmer productivity by not requiring sequential code rewriting to exploit parallelism; and (b) offer efficient programming abstractions to avoid the need for the programmer to work on low-level or architecture dependent code.

In SPar, the parallelism is expressed by means of C++ attributes. The definition of C++ attributes were added in C++11 standard, allowing programmers to provide additional information on the source code. Attributes are inserted between double square brackets and can be used to annotate types, classes, code blocks, and may be put almost anywhere on the code [GDTF17]. An example of annotation which are part of C++ standard is the `[[deprecated]]` attribute, which can be used to identify names and entities whose use are discouraged [Int17]. Leveraging the C++11 attributes to identify parallel code blocks permits the applications developer to be able to exploit the parallelism without learning a new syntax.

The SPar's focus is on the parallel patterns Pipeline and Farm, given that these patterns are better suited to the stream processing domain, as discussed in Section 2.1. The SPar compiler generates parallel code to the FastFlow [ADKT17] framework and leverages on its features such as task scheduling and stream ordering [GDTF17].

The sequential source code should be annotated with C++11 attributes to enable the SPar compiler to make the transformations that enable the parallel execution. There are 5 attributes in SPar syntax, named after the stream processing applications domain, categorized in identifier (ID) and auxiliary (AUX) attributes. Each SPar annotation in the source code contains exactly one ID attribute and zero or more AUX attributes.

Figure 2.5 presents a sample sequential pseudo-code annotated with SPar attributes and a high-level representation of the execution flow it represents. The communication between threads happen through non-blocking FastFlow queues.

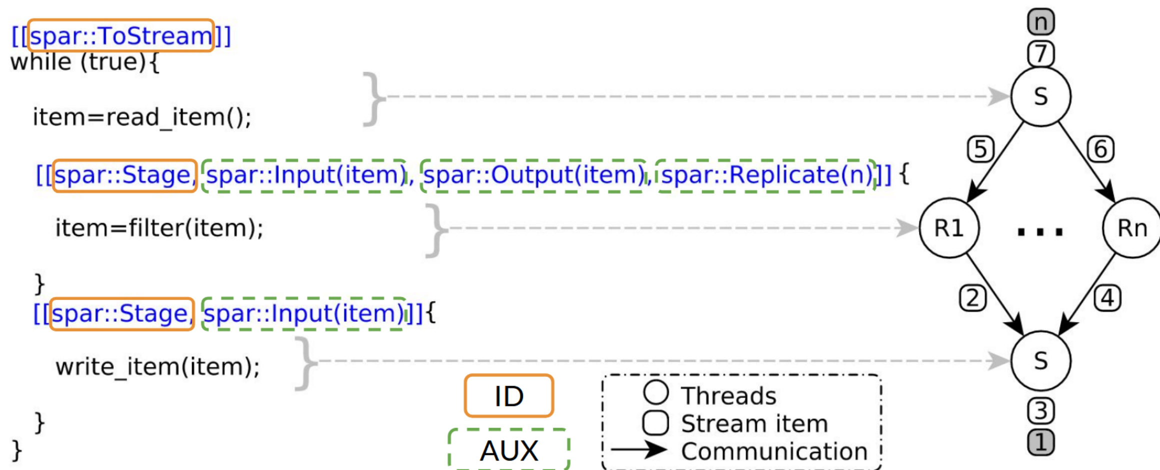


Figure 2.5: Sequential pseudo-code annotated with SPar. Adapted from [GHDF18].

The two identifier (ID) attributes are `ToStream` and `Stage`. `ToStream` identifies the code region (a compound statement or a single iteration statement such as `for` or `while`) on which stream parallelism should be employed. Inside this region the `Stage` attribute is used to identify the pipeline stages or computing phases, analogous to a assembly line. Each `ToStream` region should contain at least one `Stage` region.

There are three auxiliary (AUX) attributes: `Input`, `Output`, and `Replicate`. `Input` specifies the variables that represent the input data of the stream region (when used together with `ToStream`) or the stage region (when used together with `Stage`). `Output` specifies the variables that represent the output generated by the stream or stage region, according to the ID attribute in the same annotation. The `Replicate` attribute should only be used together with the `Stage` attribute. It specifies that the stage has no internal state and can run in parallel. Thus, increasing its degree of parallelism (i.e. the number of worker replicas). If the number of replicas is not specified, SPar uses the environment variable `SPAR_NUM_WORKERS`. The work of [Vog18] aims to provide adaptive parallelism during runtime instead of using the same parallelism degree during the entire execution.

```
1  int prime_number (int n) {
2    int total = 0;
3    [[spar::ToStream, spar::Input(total,n), spar::Output(total)]]
4    for (int i = 2; i <= n; i++) {
5      int prime = 1;
6      [[spar::Stage, spar::Input(i,prime), spar::Output(prime),
      spar::Replicate()]]
7      for (int j = 2; j < i; j++) {
8        if ( i % j == 0 ){
9          prime = 0;
10         break;
11       }
12     }
```

```
13      [[spar::Stage, spar::Input(prime), spar::Output(total)]]
14      { total = total + prime; }
15    }
16    return total;
17  }
```

Listing 2.6: Prime numbers annotated with SPar.

Listing 2.6 presents a sample application annotated with SPar which computes the quantity of prime numbers up to a predefined limit. The `ToStream` attribute in line 3 delimits the streaming region. Lines 4 and 5 are responsible for the stream management and for providing workload for the next stages. They represent the first Pipeline stage. The first `Stage` annotation (line 6) is followed by the `Replicate` auxiliary attribute, where the number specified means the degree of parallelism of this stage. The second and last `Stage` attribute appears in line 13 and collects the result of the previous stage on a reduction operation, outputting the total quantity of prime numbers found. The `Input` and `Output` attributes are used throughout the code to provide information on data items that flows from one stage to another, which is the actual task.

2.5.1 Source-to-source transformation rules

In his PhD Thesis, Griebler [Gri16] designed the original structure of the SPar language. The SPar attributes are combined in annotation schema, which trigger transformation rules in the compiler. These transformation rules are based on previous definitions.

To express the SPar definitions and transformation rules, Griebler created a particular notation: `ToStream` and `Stage` attributes are represented by T_{id} and S_{id}, where *id* represents a numeric identifier. `Input`, `Output`, and `Replicate` attributes are represented by I_i, O_i, and R_n, respectively. I_i and O_i may contain a list of typed variables a_i, and n denotes the integer number of replicas for `Replicate` argument. To denote a code block with one or more statements it is used $\square_{id}$. The scope of the sentence is denoted by $\{...\}$. The annotations that contain one identifier attribute and optionally a list of auxiliary attributes, are denoted using [[...]] [Int17].

The current definitions and transformation rules of SPar are generating the stream parallel patterns, Pipeline and Farm. These rules are implemented in the SPar compiler for transforming the annotated code into C++ code with calls to the FastFlow library, which provide classes and built-in functions for implementing these parallel patterns. Griebler uses functional semantics to define the Farm and Pipeline patterns: $farm(E, W, C)$ has arguments E (Emitter, the stream item scheduler), W (Worker, that compute stream items), and C (Collector, which gather results/stream items from the workers), where E, C, and W receive a $\square_{id}$ as argument; and $pipe(S_1, S_2, ...)$ has two or more stages, which can be $\square_{id}$ or

farm instances. The current SPar transformation rules can generate a combination of these patterns based on the annotation schema.

Currently, there are six auxiliary definitions for the transformation rules, presented in Table 2.1.

Table 2.1: Definitions for transformation rules from [GDTF17].

D_0	A *generic code block* ψ is generated for gathering results when the last $\Box$ is annotated with S containing in its attribute list R_n and O_i.
D_1	A $\Box$ can be the argument of a *pipe* pattern stage, or of a E or C in a *farm*, when its S annotation list does not contain the R_n attribute.
D_2	A $\Box$ becomes an argument of W in a *farm* pattern when it is annotated with S containing an R_n attribute.
D_3	A T becomes a *farm* pattern when the first S annotation contains R_n in the attribute list of two S at maximum.
D_4	A T becomes a *pipe* pattern when the first S does not have R_n in the attribute list or when there are more than two S annotations.
D_5	A *farm* pattern becomes a stage for the *pipe* pattern when D_3 does not apply and $\Box$ is annotated with S that contains R_n in the attribute list.

Applying D_1, D_2 and D_3 from Table 2.1, we can introduce a basic transformation rule from T to *farm* in Rule 2.1.

$$[[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}\} \Rightarrow farm(E(\Box_0),\ W(\Box_1)) \tag{2.1}$$

In Rule 2.1, $\Box_0$ and $\Box_1$ are transformed into the *farm*'s emitter and worker stages, respectively. Similarly to demonstrate D_0, we introduce Rule 2.2, which creates a generic collector stage ψ since the O_i attribute is present in the last $\Box$.

$$\begin{gathered} [[T_0]]\{\Box_0, [[S_0, O_i, R_n]]\{\Box_1\}\} \\ \Downarrow \\ farm(E(\Box_0),\ W(\Box_1),\ C(\psi)) \end{gathered} \tag{2.2}$$

The case where D_0 can not be applied and the C stage of the *farm* is generated from a $\Box$, it is demonstrated by Rule 2.3. This transformation rule applies on Listing 2.6.

$$\begin{gathered} [[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}, [[S_1]]\{\Box_2\}\} \\ \Downarrow \\ farm(E(\Box_0),\ W(\Box_1),\ C(\Box_2)) \end{gathered} \tag{2.3}$$

The *Pipeline* pattern is generated by applying D_4, which is demonstrated by Rule 2.4. The absence of R_n attribute in the stage annotation differs this case from the previous one, turning T into a *pipe* instead of a *farm*.

$$[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}\} \Rightarrow pipe(\Box_0,\ \Box_1) \tag{2.4}$$

Finally, D_5 allows for pattern composition, where the *farm* is a stage of a *pipe*. Rule 2.5 details how the $\Box$ are mapped into the patterns.

$$\begin{gathered} [[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}, [[S_1, R_n]]\{\Box_2\}\} \\ \Downarrow \\ pipe(\Box_0,\ farm(E(\Box_1),\ W(\Box_2))) \end{gathered} \tag{2.5}$$

The definitions D_0 and D_1 may also apply in Rule 2.5 if there is O_i in S_1 attribute list or another S after $\Box_2$. For the sake of simplicity, we will not provide further details, however, other more sophisticated rules can be found in [GDTF17, Gri16].

2.5.2 SPar compiler

The SPar compiler was generated from CINCLE (Compiler Infrastructure for New C/C++ Language Extensions) [Gri16]. The CINCLE provides tools for C++ source code analysis, with support up to C++14 standard. It also offers an API for performing source-to-source transformations in the resulting AST (Abstract Syntax Tree).

Figure 2.6[3] illustrates the compilation flow of the SPar compiler: (a) the compiler's input is a C++ source code, optionally annotated with the SPar attributes; (b) the GCC (GNU Compiler Collection) compiler for C++ is leveraged to perform C++ semantics analysis of the input source code; (c) the code is scanned and parsed into an AST; (d) the transformation rules of SPar are applied in the AST and generate a transformed AST; (e) the transformed AST is then assembled into C++ code and compiled using the GCC compiler; (f) the output of the compiler is a binary executable file.

The SPar compiler supports three flags that allows the programmer to control runtime behaviors in the stream processing. They work like switches passed in the compiler call:

- `spar_ordered`: this flag defines that the ordering of the input items must be respected in the output stream. The Pipeline pattern (with potentially stateful operators) naturally

[3] Icons made by Pixel perfect from www.flaticon.com

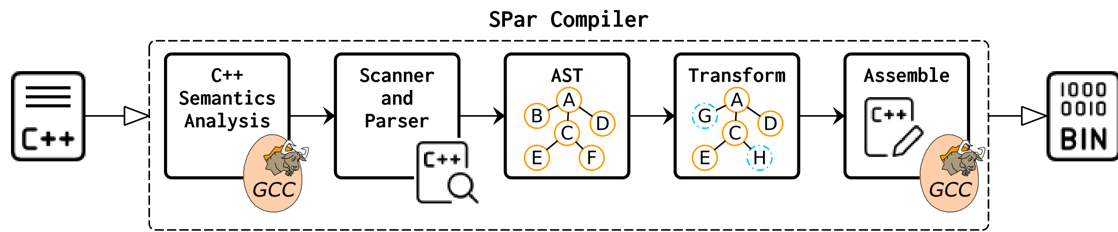


Figure 2.6: High-level representation of SPar compiler flow.

provides ordering guarantees, however, when using the Farm pattern with replicated stateless operators, the output items may arrive in the collector at different orderings. When this flag is used, SPar generates FastFlow's `ff_OFarm` class to ensure the ordering of the items at the end of the replicated stage.

- `spar_ondemand`: this flag sets the use of the FastFlow's on-demand scheduler. Without this flag, SPar uses the default dynamic round-robin scheduler of FastFlow [ADKT17], on which the emitters schedules stream items to workers in a round-robin fashion, skipping worker with full input queue (the default size is 512 items). The `spar_ondemand` flag activates FastFlow's on-demand (or "auto-scheduling") policy, on which the workers "ask" for a stream item to be computed rather than (passively) accepting items sent by the emitter [Tor15]. In practice, the worker's input queue size is reduced to 1 [ADKT17], thus workers with higher throughput receive more tasks by the emitter.

- `spar_blocking`: By default, SPar uses the non-blocking behavior of FastFlow nodes. This means that the nodes keep active in a busy waiting loop when there are no input items, and allows for quick reaction when a data item arrives in the input queue. In this mode, the number of nodes should not exceed the number of logical CPU cores. The `spar_blocking` flag switches to the blocking mode, on which the nodes blocks on the input queue until a new data item arrives. This saves energy by not keeping the processor busy but may induce latency.

3. RELATED WORK

Most developers of parallel applications currently exploit multi-core parallelism using industry standard tools such as OpenMP. However, to exploit GPU parallelism most developers use CUDA or OpenCL, which requires the programmer to learn a new lower-level language and understand hardware details in order to achieve satisfied performance using the specialized hardware. Moreover, the programmer must exploit multi-core and many-core parallelism combined to achieve good performance results. The need for a unified programming model between CPU and GPU architectures is further highlighted by the burdens of GPGPU discussed in Section 2.4.

The AMD's Heterogeneous System Architecture (HSA) standard [HSA18a] builds upon the OpenCL idea of a unified programming model for heterogeneous systems. HSA is a system architecture that defines low-level requirements for hardware and compilers to support a unified programming model, which is represented by the Heterogeneous System Architecture Intermediate Language (HSAIL) [HSA18b, HSA18c]. HSAIL is a virtual machine and intermediate language that abstracts the native instruction set and allows the execution of a single program in a larger range of HSA platforms [HSA18b]. However, there has been little movement by compiler developers to support HSAIL.

Parallel programming models for heterogeneous many-core architectures may be categorized according to the abstraction level they provide. [FHTW20] categorizes these models in two broad families: (a) the low-level programming models, which are typically bound to the hardware architecture details and expose most of these details to the programmer, offering increased performance but lower programmability; and (b) the high-level programming models, which hide most hardware architecture details by raising the abstraction level, usually using some low-level model as backend, and trading performance for increased programmability.

Since the level of the programming abstractions provided by the high-level programming models varies greatly, this family of models is further classified by [FHTW20] in five types, based on the technology used to provide the programming abstractions. Ordered from lower to higher abstraction level, these five types are: (I) based on modern C++ features. Examples are Khronos' SYCL [Khr20c], Microsoft's C++AMP [GM12], VexCL [DARG13], and PACXX [HG14]; (II) based on structured parallel programming (skeletons or parallel patterns). Tools like SkelCL [SKG11], SkePU [ELK18], and FastFlow [ADKT17] are in this category; (III) based on the C++ Standard Template Library (STL) API, such as Thrust [NVI19], Boost.Compute [Lut15, Szu16], Bolt [AMD14], and Kokkos [CTS14]; (IV) based on code annotations. The works in this category include efforts from industry, such as OpenMP [Ope18] and OpenACC [Ope15a, Ope17], as well as efforts from academia, such as *hi*CUDA [HA11], XscalableMP-ACC [LTO+12], AHP [PCR12], and HDAr-

ray [CKM19]; (V) based on domain-specific concepts. Some examples are Halide [RKBA+13], StreamIt [TKA02, UGT09, HSW+11], Single Assignment C (SAC) [GTS11], ArrayFire [MYM+12], and Nebo [EMBS17].

These five types does not have rigid boundaries. For example, Thrust and Boost.Compute provide STL-like APIs (III), however, they offer high-level patterns such as Map (under the name of `transform`), Reduce, Scan, and Gather. Therefore, they can also be classified as structured parallel programming APIs (II). Our work builds upon the SPar language, which is a domain-specific language (V) that employs C++11 code annotations (IV) to express parallelism in sequential source code. Therefore, this classification of high-level programming models are a *best effort* rather than definitive labels.

The next two sections present works related to this study in two main aspects. First, Section 3.1 present structured parallel programming APIs for GPU programming, which are related to our novel library, (GSPARLIB). Second, Section 3.2 presents annotation-based approaches, which are works related to our extension to the SPar language targeting combined stream and data parallelism in heterogeneous computer architectures composed of multi-core CPUs and many-core GPUs. Since SPar is a domain-specific language embedded in C++ that employs C++11 attributes, it does not make sense to compare it to other DSLs that require programmers to rewrite sequential code in order to exploit parallelism using high-level abstractions. Therefore, we compare it with tools that use the same mechanism (code annotations) to expose parallelism in the code.

3.1 Structured parallel programming APIs for GPU programming

In this section, we approach works related to our novel structured parallel programming library for GPGPU (GSPARLIB). We selected studies that offer structured programming APIs to exploit GPU parallelism. Therefore, these tools present an abstraction layer above the tools presented in Section 2.4, which only provide procedural APIs.

SkelCL [SKG11] is a library focused in data parallelism algorithmic skeletons that supports common data-parallel patterns: Map, Reduce, Zip (a special case of Gather [MRR12]), Scan, Stencil, and the custom Allpairs and MapOverlap skeletons. It allows programmers to exploit GPU parallelism using C++ by generating code for OpenCL kernels. The usage of skeletons in SkelCL happens through the use of predefined classes, which receive the code of customization functions as plain string parameters. These customization functions are combined with boilerplate code, which are predefined inside the skeleton to generate the final OpenCL code. We use a similar approach in GSPARLIB, which will be discussed in more details in Chapter 4. The final OpenCL kernel code is then compiled at runtime and stored on disk, avoiding recompilation when the kernel is reused. We could not find any information about thread-safety of SkelCL in their papers or the library's documentation. Moreover, the

works on the library seems to have ceased some years ago (the last paper[1] was published in 2014 and the last commit in the library's Git repository[2] is from 2016).

Thrust [NVI19] is a template library launched in 2009 by NVIDIA and open sourced under Apache 2.0 license. It offers a structured programming alternative to exploit GPU parallelism using CUDA. The memory transfers between the main (host) memory and device memory occurs mainly by means of two classes from the `thrust` namespace: `host_vector` and `device_vector`. Since Thrust is integrated with CUDA, kernels can be defined as plain C++ functors annotated with CUDA's `__device__` keyword, which are converted to device kernel code by NVIDIA's compiler, `nvcc`. The kernel can be launched by passing it as argument to one of the C++ STL-like functions provided by Thrust, such as `thrust::transform` (which works just like the *map* pattern) and `thrust::reduce`. It also supports other parallel patterns, such as *stencil* (`thrust::partition` combined with `thrust::transform`) and *scan* (`thrust::inclusive_scan` and `thrust::exclusive_scan`). However, Thrust does not offer any support for stream parallel patterns (e.g. Pipeline and Farm). We could not find any information about thread-safety capabilities of Thrust library in their published documentation. Nonetheless, in our tests the library worked well in a multi-threaded environment.

Boost.Compute [Lut15] is a header-only library that leverages OpenCL for GPGPU. It was developed by Kyle Lutz [Lut15] and Jakub Szuppe [Szu16] and became part of the Boost set of libraries in 2016. The library provides an API composed of two layers: a low-level API that is a thin C++ wrapper over OpenCL; and a high-level API that offers STL-like methods such as `transform`, `gather`, `reduce`, and `sort` [Szu16], which are all in the namespace `boost::compute`. Custom kernels can be defined in special lambda expressions (using special placeholders to access the arguments: `_1`, `_2` ... `_N`) or by using the `BOOST_COMPUTE_FUNCTION` and `BOOST_COMPUTE_CLOSURE` macros.

Boost.Compute offers an opt-in thread-safety feature[3], which can be enabled by defining the `BOOST_COMPUTE_THREAD_SAFE` macro. However, its thread-safe version leverages the Boost.Thread library and thus requires linking it during the compilation phase. Boost.Compute's high-level API resembles the Thrust library, since both aims at STL-like methods. Like Thrust, Boost.Compute does not offer any support for stream parallel patterns.

FastFlow [ADM+09, ADKT17] is a framework to stream parallelism based on algorithmic skeletons. It was created in 2009 by members of the University of Pisa and University of Torino, Italy. FastFlow offers a library of templates in C++ to express parallelism in stream processing applications through the use of the Pipeline and Farm patterns. GPGPU support was published in 2015, starting with the *stencil-reduce* pattern [APD+15], which was subsequently extended [ADD+18] under the name of *loop-of-stencil-reduce*. This extended pattern is general enough to allow the implementation of Map, Reduce, Stencil and its combinations.

[1] https://skelcl.github.io/index.html#publications
[2] https://github.com/skelcl/skelcl
[3] https://www.boost.org/doc/libs/1_74_0/libs/compute/doc/html/boost_compute/faq.html#boost_compute.faq.is_boost_compute_thread_safe_

Our approach is very similar to FastFlow's. However, we provide a different abstraction level. To exploit the GPU parallelism in FastFlow using the *loop-of-stencil-reduce*, the programmer needs to provide the CUDA or OpenCL kernel code. Even though FastFlow offers some macros to ease the generation of the boilerplate code in the GPU kernel, they are limited. For example, the macros `FFMAPFUNC` to `FFMAPFUNC6` allow the user to generate a simple CUDA kernel with one to six arguments. If the programmer needs any different number of parameters in the kernel, they cannot use these auxiliary macros. Other limitation is that these macros must be called in the global scope since they generate a C structure containing the device function. The availability and functionality of these macros for generating kernel code varies between the CUDA and OpenCL backends, therefore, FastFlow does not provide a true driver-agnostic API. As a library, FastFlow does not offer the transformation of C++ code into CUDA nor OpenCL, but instead eases the management and execution of programmer-defined kernels [ADD+18]. The current GSParLib version also does not automatically transform C++ code into lower-level languages, but it abstracts most of the common differences by providing helper functions (in the lower-level API) and automatically generating boilerplate code (in the higher-level API). Moreover, when using the FastFlow's CUDA backend, the code must be compiled with the `nvcc` compiler. In GSParLib, the code may be compiled with any C++ compiler since we use the lower-level CUDA Driver API.

Another effort in heterogeneous parallel programming using parallel patterns is represented by SkePU [ELK18]. SkePU is a skeleton-based programming framework developed using C++ by members of the Linköping University, Sweden. It is composed of a source-to-source compiler tool and a parallel runtime. It contains skeletons for the parallel patterns Map, Reduce, Stencil (which is named `MapOverlap` in the SkePU context), Scan and combinations of them, including MapReduce. It also provides a `Call` skeleton which does not encode a computational pattern, but instead serves as a versatile skeleton because it allows the implementation of arbitrary computational patterns [EK20]. Figure 3.1 presents an overview of SkePU 2 compilation process.

Similar to SPar, the SkePU source program may be compiled with any C++11 compiler, which outputs a sequential executable. Alternatively, using the SkePU source-to-source compiler on the SkePU source program, it is possible to obtain a parallel code, which in turn can be compiled to generate a parallel executable. This compiler generates OpenMP code for CPU parallelism, MPI, and StarPU code for cluster execution, and both CUDA and OpenCL code for GPU parallelism, which can be compiled using `nvcc` and `g++` compilers, respectively, to produce the parallel program. A preview of SkePU 3 was made available during the development of our work, but there were no paper published addressing this new version at the time of this writing. Although SkePU supports both multi-core CPU and many-core GPU parallelism, the code generated by the SkePU compiler cannot be safely integrated with other multi-threaded tools (i.e. it is not thread-safe). We confirmed that this limitation also affects SkePU 3.

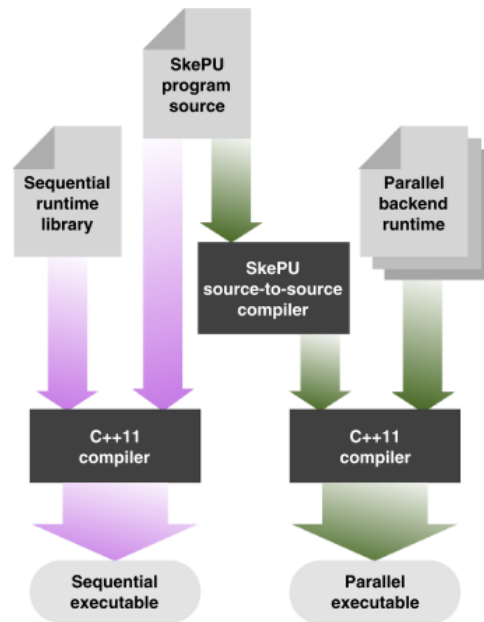


Figure 3.1: SkePU 2 compiler chain. Extracted from [ELK18].

Table 3.1 presents a summary of the related works and an overview of our work with respect to the state-of-the-art structured parallel programming APIs for GPGPU. We sort the works in the order on which the GPGPU support was developed, i.e. the first work is the older. FastFlow, for example, was created in 2009 [ADM+09] but GPGPU support was published in 2015 [APD+15]. The *Runtime support* columns (*CUDA* and *OpenCL*) indicate which API are supported as the runtime backend. The *Runtime abstraction* column evaluates if the runtime boilerplate code is abstracted from the programmer, i.e. if the programmer is relieved from providing the entire kernel code in a lower level language. The *Map and Reduce* column evaluates whether the software supports these patterns, which are commonly used to exploit data parallelism. Although it is possible to build almost any parallel pattern in most of the tools since they offer a lower-level API, we only check-marked the tools with high-level built-in support for such patterns.

The *Batch* column indicates if the API provide any support for merging multiple kernel calls into a single call, which is a useful feature for integration with stream processing applications that aims at the batching optimization discussed in Section 2.1. The *Thread-safe* column indicates whether the software can be safely executed in a multi-threaded environment and take advantage of multiple CPU cores. Finally, the *Native containers* column indicates if the tool supports performing GPGPU using the native data containers of C++ (pointers or STL containers). A $\times$ mark in this column indicates tools that require the use of custom data containers. On the one hand, providing custom data containers may ease the programmability of the language, since they are usually used to provide automatic data copies between the main memory and the GPU memory. On the other hand, if the tool requires the use of such custom containers, the code refactoring of sequential applications that use the native containers can be cumbersome.

Table 3.1: Related Work of structured parallel programming APIs for GPGPU.

Ref	*Name*	*Objective*	*Runtime support*		*Runtime abstraction*	*Map and Reduce*	*Batch*	*Thread-safe*	*Native containers*
			CUDA	*OpenCL*					
[NVI19]	Thrust	Flexible and high-level interface for GPU programming to enhance developer productivity	✓	×	✓	✓	×	✓	×
[SKG11]	SkelCL	Library with a high-level approach for GPU programming	×	✓	✓	✓	×	?	×
[Lut15]	Boost.Compute	GPU library for C++ based on OpenCL.	×	✓	✓	✓	×	✓	×
[APD+15]	FastFlow	Simplifying the implementation of data-parallel programs on heterogeneous multi-core platforms	✓	✓	×	✓	×	✓	✓
[ELK18]	SkePU 2	Skeleton programming framework for heterogeneous parallel systems	✓	✓	✓	✓	×	×	×
Our work	**GSPARLIB**	**Structured programming API for data parallelism in stream processing applications**	✓	✓	✓	✓	✓	✓	✓

3.2 Annotation-based approaches

In this section we approach works related to our extension to the SPar language targeting data parallelism in heterogeneous computer architectures composed of multi-core CPUs and many-core GPUs. Therefore, we selected studies providing a high level programming abstraction, aiming at automatic and semi-automatic parallelism for GPUs.

Given the burdens of GPU parallel programming, discussed in details in Section 2.3, tools for automatically inserting code for GPU APIs into sequential C or C++ code without programmer intervention is of significant interest. Both [BRS10] and [VJC+13] presents automatic code translation systems which leverage polyhedral compiler transformation to convert sequential C code into CUDA. However, they are limited to affine regions of code, which are composed of expressions for accessing arrays in a sequential manner inside a loop [Ben18]. DawnCC [MGA+17] is a tool aimed at automatically generating OpenACC and OpenMP annotations by leveraging symbolic range analysis. This technique allows them to handle nonaffine regions of code, however, polyhedral-based tools are able to perform more aggressive transformation in loops, such as tiling and fission [MGA+17].

Fully automatic parallelization of sequential C and C++ code is the ultimate goal of parallel programming abstractions. It has the potential to improve the productivity of the application programmers by providing portability and abstracting the underlying hardware architecture completely. However, current technologies for code analysis and transformation are not able to make efficient parallelism choices [EAB+20]. They are usually specific in either their ability to introduce parallelism, the code to which they can be applied, or both [BJB+20]. Some industry experts even believe that such a compiler for automatic parallelism will never exist [Sco12]. For example, in our tests using the online version of the DawnCC compiler[4], it was not able to add annotations in the C version of any of the applications we used for our performance tests in Sections 4.7 and 5.6.

Alternatively, annotation-based solutions for parallelism exploitation permit the users to indicate parallel regions to the compiler without requiring significant changes to the sequential source code. OpenMP [Ope18] offer `pragma`-based annotations to allow the programmer to express parallelism in the code. It is the *de facto* industry standard for multi-core CPU parallelism, used in 7 out of the 10 top supercomputers worldwide as reported in November 2019 [FHTW20]. The OpenMP language committee started working towards heterogeneous parallelism capabilities in the version 4.0 of the OpenMP specification and improved their support for offloading in version 4.5 [Ope15b]. However, the programmer still needs to know architecture details to achieve good performance [Lar18].

OpenACC [Ope15a, Ope17], which was already presented in Section 2.4.3, is another annotation-based tool developed by the industry that uses `pragma`s for annotating

[4]http://cuda.dcc.ufmg.br/dawn/index.php

code suitable for GPU offloading. Although it leverages high-level annotations, the OpenACC language is composed of lower-level concepts such as `worker`, `vector`, and `gang`. Therefore, it also requires the programmer to know hardware details in order to properly exploit GPU parallelism.

There are also `pragma`-based tools developed by the academia, such as *hi*CUDA [HA09, HA11] and XscalableMP-ACC [LTO+12]. *hi*CUDA offer the `hicuda pragma` to specify GPGPU in source code. It offers clauses to identify GPU kernels (`kernel`), define the GPU grid size to the kernel invocation (`tblock` and `thread`), manage GPU memory (`alloc` and `free`), copy data (`copyin` and `copyout`), and define optimizations (`loop_partition`, `over_tblock`, and `over_thread`). XscalableMP (XMP) [Xca18] is a parallel PGAS (Partitioned Global Address Space) programming language focused in distributed parallelism. It provides high-level `pragma` annotations and shares many core ideas with the OpenMP standard. XscalableMP-ACC (XMP-ACC) [LTO+12] is an extension to the XMP language that adds the `acc` clause to identify regions suitable to GPU offloading, which are transformed into a CUDA kernel by the compiler. Besides its similarity in naming, XMP-ACC does not have any relation with OpenACC. Both *hi*CUDA and XMP-ACC requires the programmer to explicitly manage memory movements and lack a high-level abstraction of the GPU architecture. Moreover, they are tied to NVIDIA boards due to their CUDA backend.

More recently, Nakao et al. [NMS+14] published XcalableACC (XACC), another extension to the XMP language based on the OpenACC model. This extension was implemented in a custom compiler, which translate the extended set of directives into XMP runtime functions and OpenACC directives, which are then fed into an OpenACC compiler. This extension seems to have been officially introduced into the XMP ecosystem[5], however, it remains as a separate language and it is not signed by the XcalableMP Specification Working Group [RIK17]. However, the language is still very much verbose and not intuitive from an application programmer perspective. An example is that it requires programmers to indicate memory allocations (`xmp shadow` and `reflect`) and data copies (`xmp gmove`), both into the GPU (`acc data copy`) and out from the GPU (`acc update host`).

The Automatic Heterogeneous Pipelining (AHP) framework [PCR12] focuses on identifying the pipeline stages, mapping them into the available processing units (PUs) of the heterogeneous system, and scheduling their execution. Although the AHP language resembles SPar, we focus on providing efficient and high-level abstractions decoupled from the underlying hardware to offer the opportunity of application programmers to exploit the parallelism of heterogeneous systems. On the other hand, AHP expects the programmer to provide hand-written optimized code for the available PUs [PCR12], which actually increases the programming effort since the user must provide optimized variants for all available PUs that can possibly be used by AHP during runtime.

[5]https://xcalablemp.org/XACC.html

All the aforementioned solutions leverage `pragma` preprocessing directives to provide annotations for heterogeneous parallelism. The C++ attributes introduced in the C++11 standard offer additional flexibility since they can be inserted almost anywhere in the code, and does not require any preprocessing [GDTF17, DGS+16]. Moreover, they are part of the C++ grammar [Int17], unlike the `pragma` mechanism.

Parallelism exploitation by means of C++11 attributes were promoted by REPARA (Reengineering and Enabling Performance And poweR of Applications) [DGS+16]. REPARA was an European project that ran between 2013 and 2016, trying to promote the use of C++11 attributes to express stream and data parallelism by annotating parallel patterns in sequential source code [DDMT18]. It included annotations for the parallel patterns Pipeline (`rpr::pipeline`), Farm (`rpr::farm`), and Map (`rpr:map`). The programmer may annotate parallel regions using the `rpr::kernel` attribute (which does not allow nesting), and specify the input (`rpr:in`) and output (`rpr:out`) parameters. REPARA also offers the `rpr:async` attribute to signal asynchronous kernel execution and `rpr:sync` to wait for an asynchronous kernel. They also proposed the `rpr:target` attribute to indicate heterogeneous computer architectures such as GPU and FPGA, but the support for heterogeneous parallelism is left as future work [DDMT18]. Contrary to SPar, the REPARA project never assembled a language interpreter and source-to-source compiler to implement the proposed transformation rules, being a theoretical work.

Table 3.2 summarizes the differences between the aforementioned related works and our current work. The *GPU backend* column specifies the underlying lower-level API that is employed to exploit GPU parallelism. The REPARA project does not supports GPU parallelism and therefore is left unfilled. The *Annotation mechanism* column specifies which mechanism is employed to provide code annotation (`pragma` or C++11 attributes). The *Supported parallelism* columns identify which parallelism is supported by the tools: *Multi-core*, *GPU*, and *Distributed* (for distributed memory architectures). We only check-marked tools that support these parallelism levels to be exploited simultaneously. For example, the PGI 15.10 compiler for OpenACC supports multi-core parallelism, however, it does not allow to combine it with many-core GPU parallelism, therefore, it is marked with $\times$. Also, XMP recommends the use of OpenMP for parallelism inside each node of the distributed system [Xca18]. Therefore, it does not support CPU multi-core parallelism by itself. Finally, the *Architecture abstraction* column identify the works which successfully abstract the underlying hardware architecture from the programmer's perspective.

Table 3.2: Related Work of annotation-based APIs for heterogeneous parallelism.

Ref	*Name*	*Objective*	*GPU backend*	*Annotation mechanism*	*Supported parallelism*			*Architecture abstraction*
					Multi-core	*GPU*	*Distributed*	
[Ope18]	OpenMP	Provide a model for parallel programming that is portable across architectures from different vendors	Compiler-dependent[a]	`pragma`	✓	✓	×	×
[Ope17]	OpenACC	Provide a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators	Compiler-dependent	`pragma`	×	✓	×	×
[HA11]	*hi*CUDA	Provide a high-level directive-based language for CUDA programming	CUDA	`pragma`	×	✓	×	×
[LTO+12]	XMP-ACC	Provide a productive parallel programming model for multi-node GPU clusters	CUDA	`pragma`	×	×	✓	×
[RIK17]	XACC	Provide a parallel programming model for accelerated clusters which are distributed memory systems equipped with accelerators	OpenACC	`pragma`	×	✓	✓	×
[PCR12]	AHP	Provide an automatic framework for the creation of heterogeneous pipelines from annotated non-pipelined program specification	CUDA	`pragma`	✓	✓	×	×
[DGS+16]	REPARA	Promote the use of C++11 attributes to express parallelism	–	C++11 attributes	×	×	×	×
Our work	**SPar**	**Design efficient and high-level parallel programming abstractions for expressing parallelism on stream processing applications targeting heterogeneous parallel architectures**	**GSPARLIB**	**C++11 attributes**	✓[b]	✓	✓[c]	✓

[a] GCC compiler supports compiling OpenMP annotations to NVIDIA PTX, AMD HSAIL, AMD GCN (Graphics Core Next), and Intel MIC [GCC20].
[b] SPar's support for multi-core CPU was originally implemented in [GDTF17]. We implemented the batching optimization for exploiting GPU parallelism.
[c] SPar's support for distributed memory architectures was implemented in [GF17, Pie20].

4. GSPARLIB: UNIFIED GPU LIBRARY FOR STREAM PARALLELISM

This chapter presents our novel structured parallel programming library for GPU, GSPARLIB. It provides a unified interface and driver-agnostic runtime, wrapping both CUDA and OpenCL languages in a fluent C++ skeleton-based API while enabling different GPU vendors. Our interface is unified because the programmer can express parallelism in the same way using generic patterns while using different low-level GPU programming models (CUDA and OpenCL). Also, the library is driver-agnostic because enables executing on different GPU driver vendors without requiring code refactoring. If the programmer eventually wants to change the target GPU driver vendor, he only needs to recompile the program specifying a compiler flag. Since it focuses in stream processing applications, some key features of GSPARLIB are thread-safety and support for batching stream items. Moreover, it uses the native C++ data containers to ease the process of porting existing (sequential) applications.

We discuss the motivation behind the decision to develop GSPARLIB with a comparison to other runtime alternatives in Section 4.1. Section 4.2 presents concise guidelines on how to use GSPARLIB in a high-level of abstraction, explaining how to apply it to existing applications. The main choices made during the library design related to the abstractions it provides for the domain of GPU programming are discussed in Section 4.3. Then, we present the two layers of GSPARLIB API: the lower-level Driver API (Section 4.4) and the higher-level Pattern API (Section 4.5). In Section 4.6 we discuss the programmability of GSPARLIB compared to other GPU programming alternatives. Finally, Section 4.7 presents a set of experiments comparing GSPARLIB's performance with the state-of-the-art tools discussed in Section 4.1, and Section 4.8 present our final remarks and concludes this chapter.

4.1 Motivation

The *de facto* standard APIs for GPU programming are CUDA and OpenCL [FHTW20], which offer only procedural programming without any support for structured programming. Even StarPU, which is a runtime system built on top of CUDA and OpenCL (discussed in Section 2.4.4) does not offer any structured programming API. It is a new task-based programming model. We instead seek a structured programming API to exploit GPU parallelism in stream processing applications, abstracting CUDA and OpenCL programming models and different GPU drivers vendors. We discussed GPU programming interfaces with structured parallel programming support in Section 3.1, namely NVIDIA's Thrust, SkelCL, *Boost.Compute*, FastFlow, and SkePU.

Both *Boost.Compute* [Lut15] and NVIDIA's Thrust library [NVI19] are widely used with an STL-like structured programming API. However, we observed a few drawbacks when considering them as a SPar runtime. First, we seek a unified interface for GPU parallelism abstracting CUDA and OpenCL programming model differences. A second downside is that both *Boost.Compute* and Thrust requires that most of the interactions between the host and device processors, such as memory transfers, must be done manually by the programmer. We want to alleviate this work from the compiler. A third disadvantage is that the *Boost.Compute* and Thrust requires issuing OpenCL or CUDA function calls to perform specific tasks such as asynchronous memory copies.

SkelCL [SKG11] uses OpenCL for GPU parallelism exploitation and thus also lacks CUDA support. Moreover, SkelCL is not actively developed anymore since the last paper was published in 2014 and the last commit in its GitHub repository[1] is from 2016. SkePU [ELK18] supports CPU parallelism using OpenMP, however, the SkePU skeletons themselves are not thread-safe, which undermines the integration with other parallel tools such as SPar with the FastFlow runtime.

FastFlow [APD+15] is an interesting alternative since it is already used by SPar for multi-core parallelism. It is also the only tool from Section 3.1 that supports GPU parallelism using native C++ containers, which is a desirable feature to ease the process of porting existing applications. However, the FastFlow's CUDA backend requires the use of the proprietary closed-source `nvcc` compiler and we want to avoid an extra compiler roundtrip in the SPar compilation process. We also want to avoid having complex rules in the compiler to generate the CUDA or OpenCL kernel code or to switch between those drivers, which is only possible if the runtime provides a unified API. FastFlow's GPU programming API is not unified. The macros provided by FastFlow to aid in the GPU kernel generation have limited applicability and would limit the applicability of SPar since it is not possible to automatically transform the sequential source code annotated with SPar attributes to the macro inputs. One of such limitations is the fact that all the macros for generating OpenCL code result in a single data vector of output, however, some kernels generate multiple data vectors of output, such as the sobel filter step of the lane detection application, which outputs the image data and also a vector of the pixels' directions (see more details in Section 5.6.3). Due to these limitations, most of the examples in the FastFlow repository[2] use hand-written kernel code. Lastly, we found out that the GPU support is not working in FastFlow's current version[3] and we are unsure about the priority of this feature for FastFlow's developers.

Providing a unified API that abstracts CUDA and OpenCL programming models is an important step towards code and performance portability. On the one hand, the CUDA driver is important for exploiting GPU parallelism in the NVIDIA chips, which is the market leader of this segment. Even though the NVIDIA chips supports the OpenCL driver, they present

[1] https://github.com/skelcl/skelcl
[2] https://github.com/fastflow/fastflow
[3] https://github.com/fastflow/fastflow/issues/38

better performance using the CUDA driver (we will discuss these performance differences in Section 5.6). Moreover, we find some issues using OpenCL on NVIDIA boards, such as lack of support for newer versions, which will be discussed in more details in Section 4.8. On the other hand, the OpenCL driver is the only way to exploit massive parallelism of GPUs from other chipmaker and even from other hardware accelerators, such as FPGAs, digital signal processors (DSPs), and embedded devices [KMSZ15, KmWH16]. Currently, GSPARLIB focuses only in GPU devices. In the future, we may leverage it for different hardware accelerators.

One big difference between writing GPU kernels in FastFlow and GSPARLIB is that FastFlow's macros are evaluated in compile-time, thus the kernel code is generated during the compilation of the application and is hard to modify it during runtime. For the CUDA approach, the nvcc compiler is used to compile the GPU kernel during compile-time, while the OpenCL kernel is generated during the compilation of the application and compiled in runtime. On the other hand, GSPARLIB generates and compiles the code of CUDA and OpenCL during runtime. Generating the code in runtime allows us to perform optimizations based on the heterogeneous computer architecture, which is not known at compile-time [AGDF20]. Moreover, macros have limited capabilities while our approach uses a more powerful tool to perform code transformation and generation because it allows complex analysis in the GPU kernel code. Therefore, we advocate that the FastFlow's decision of using macros to aid the programmer in the process of generating boilerplate kernel code is a poor design choice.

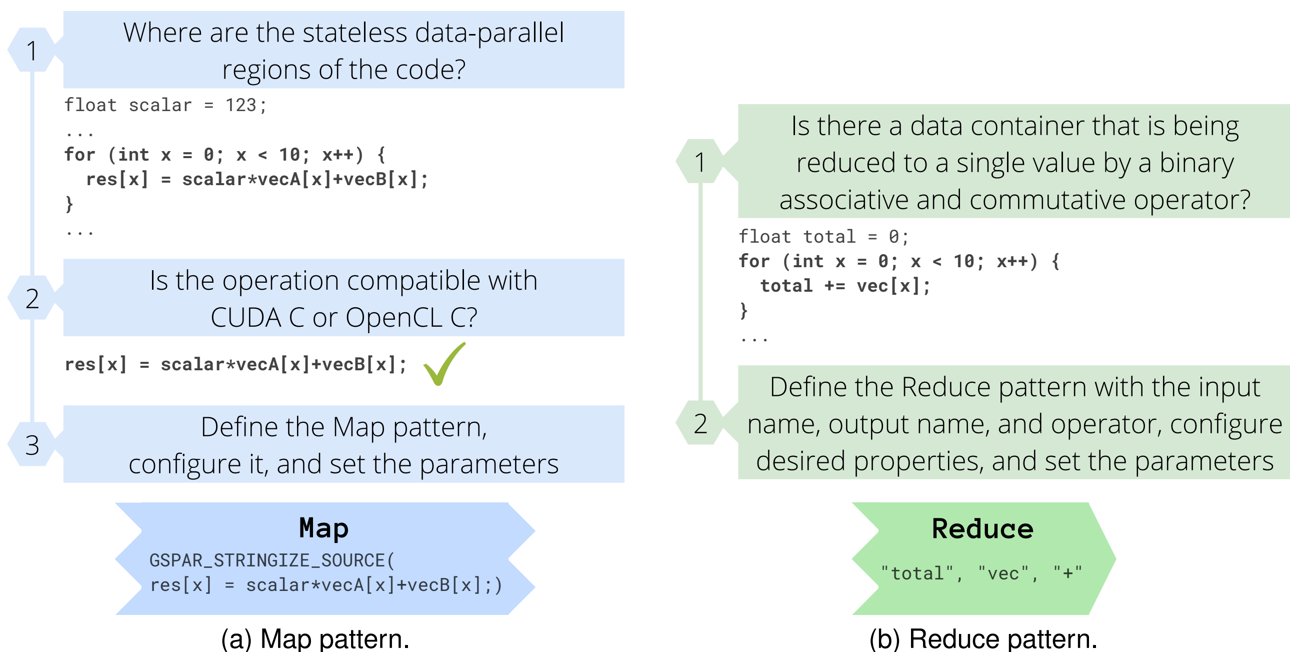
In our previous work [RSG+19], we identified the need for creating micro-batches of stream items to properly exploit many-core accelerators like GPUs. It is also desirable to be able to change the batch size during the stream execution, which allows for high-level decision making over trade-offs between latency and throughput [SRG+20]. None of the tools discussed in Section 3.1 offers support to the batching optimization in stream processing applications. Due to the all previous discussed points, we decided to develop a unified and high-level GPU library suitable for parallelism exploitation on stream processing applications.

4.2 Programming with GSPARLIB

In this section, we show how to use GSPARLIB's structured programming API for exploiting GPU's data parallelism on C++ applications. We start by defining the GSPARLIB parallel patterns. The steps of this process are illustrated in Figure 4.1. The Map pattern is highlighted with blue color in Figure 4.1a, where programmers have to follow these three steps: (1) identify where are the stateless data-parallel regions of the code to apply the Map pattern; (2) check whether the operation computed in the data-parallel region is compatible with CUDA C or OpenCL C, which is a requirement for GSPARLIB's GPU kernel code; (3) define the Map pattern object with the GPU kernel core, configure the desired properties such as batch

size and device to be used when executing the pattern, and set the pattern's parameters. In Figure 4.1a we are using the `GSPAR_STRINGIZE_SOURCE` macro to pass the GPU kernel code to the pattern, which is a code wrapper.

The Reduce pattern is highlighted with green color in Figure 4.1b and comprises two steps: (1) check if there is a data container that is being reduced to a single value by a binary associative and commutative operator; and (2) define the Reduce pattern object with the output name, input name, and operator, configure the desired properties such as the device to be used when executing the pattern, and set the pattern's parameters. In Figure 4.1b we are setting "`total`" as the output name, "`vec`" as the input name, and "+" as the operator. We are also setting the pattern to use the first GPU (which is the default).



(a) Map pattern.

(b) Reduce pattern.

Figure 4.1: Methodology for defining GSPARLIB patterns.

After defining the parallel patterns, the user can combine multiple patterns in a pattern composition. The pattern composition optimizes the combined patterns and executes them sequentially (future versions may support different execution flows). The most common pattern flow is a sequence of Maps followed by one or more Reduces. Nonetheless, currently GSPARLIB allows any combination of patterns to be combined and does not enforces special ordering.

These steps for defining the patterns and compositions can be performed during the application's warm-up phase, outside any loop of computation. Optionally, after these steps, the programmer may include a call to `compile` the patterns before the computation. Otherwise, the compilation of the pattern will occur automatically when the `run` method is called for the first time.

Figure 4.2 illustrates how to apply these steps in a simple sequential vector sum function. We highlighted in blue color the GSPARLIB code. The first step is to identify the

stateless data-parallel regions of the code, which is the for loop that performs the vector summation. In the second step we check if the operation (highlighted by a dashed orange box) is compatible with CUDA or OpenCL GPU kernel code. This operation is compatible with both backends. In the third step we define the pattern itself with the operation from the second step. In this case, right after defining the pattern, we set the parameters and run it using the `size` variable.

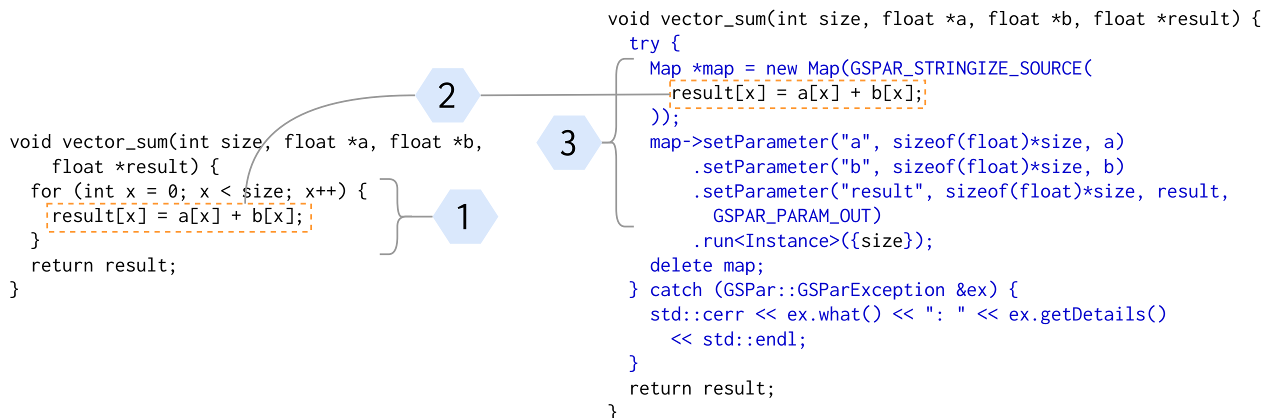


Figure 4.2: Applying GSPARLIB in a sequential code.

This simple example demonstrates the application of GSPARLIB in sequential source code, which does not involve multi-core parallelism. Now, to demonstrate the use of GSPARLIB in a stream processing application, we integrate it in a FastFlow application. It is important to highlight that GSPARLIB can be integrated with any tool focused in multi-core parallelism, such as TBB or OpenMP. We chose to integrate with FastFlow due to our familiarity with its programming model. First, Figure 4.3 shows how to modify a sequential source code using the FastFlow library to exploit stream parallelism in multi-core architectures. This code example applies a SAXPY (Single precision A X Plus Y) operation in a stream of vectors.

The right side of Figure 4.3 presents the sequential source code, on which we delimited the Farm pattern components with dashed blue boxes. We also labeled the components with filled blue boxes in the right side of the code: the `E` box marks the emitter, the `W` box marks the workers, which apply a stateless operator and thus can be performed in parallel, and the `C` box marks the collector with a stateful operator that cannot be replicated. For the sake of simplicity, we defined the `scalar` variable in the global space.

The left side of Figure 4.3 presents the code with stream parallelism using the FastFlow library. The changes with respect to the original sequential code were highlighted in green color. The first part of the code defines the `Task` structure, which represent the stream items that are sent from one stage to the next stage. Each of the Farm pattern components is defined as a class which inherits the `ff_node` or `ff_note_t` class. They define the `svc` method to perform the stage operations. The `Emitter` class obtains the input vector using the `get_next_input_vec()` functions, allocates memory for the output, and sends these

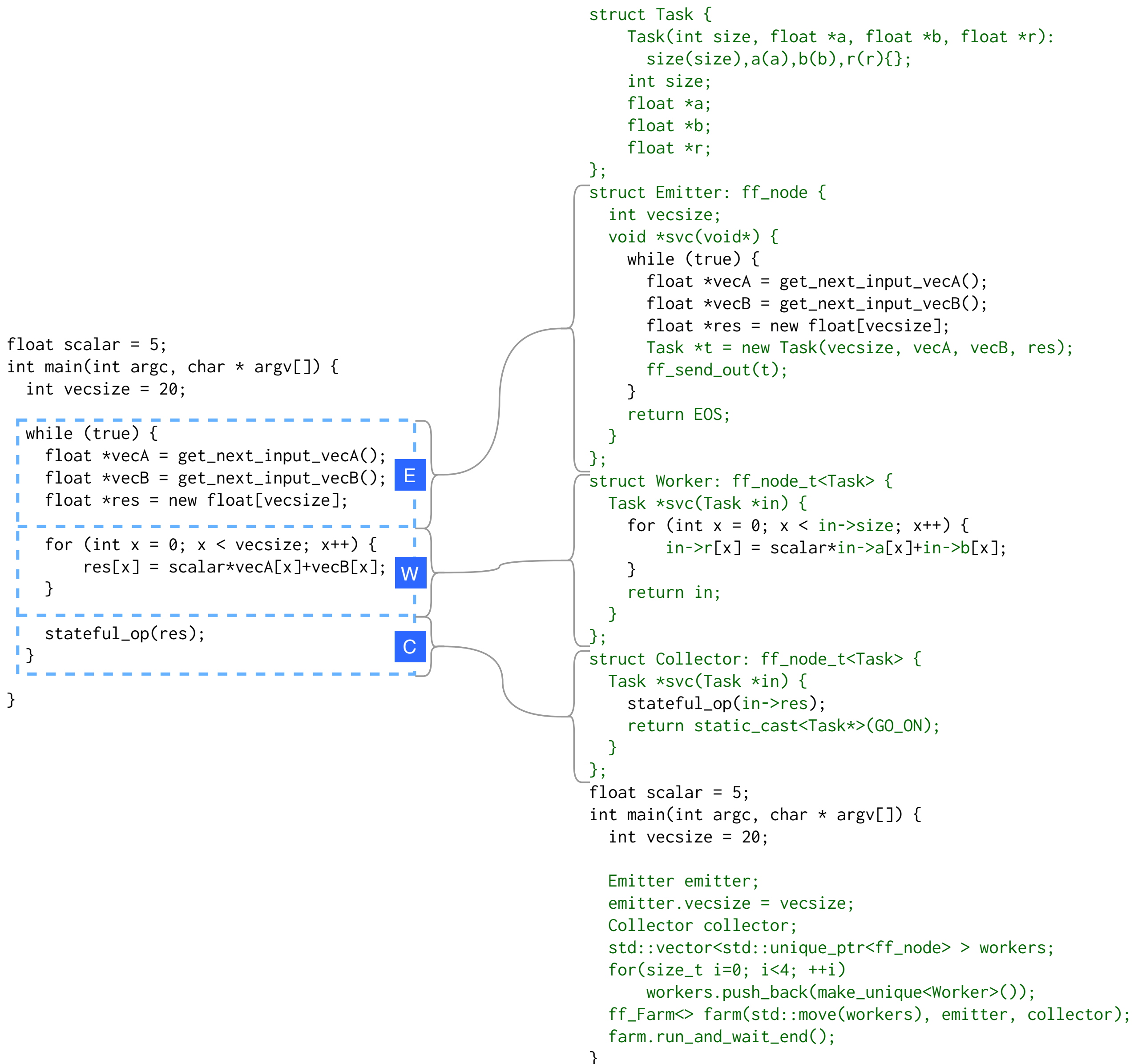


Figure 4.3: Using FastFlow library to exploit stream parallelism in sequential source code.

vectors to the workers. The `Worker` class perform the actual computation, using the vectors from the `Task` item received from the `Emitter`, and sends the item to the next stage. The `Collector` class calls the `stateful_op` function passing the result vector as argument. In the `main` function, the code was replaced by the preparation and execution of the `ff_Farm` object, which contains one instance of the `Emitter`, four replicated instances of the `Worker`, and one instance of the `Collector`.

Now we introduce data parallelism for GPU in the code with stream parallelism for multi-core architectures. Figure 4.4 demonstrates how to apply GSPARLIB together with the FastFlow code. We apply the three steps described in Figure 4.1a to define the Map pattern: (1) we identify the data-parallel stateless code section that represents the Farm's worker; (2) we restore the original (sequential) core operation, which is highlighted by a dashed

orange box, as it is compatible with both CUDA and OpenCL GPU kernel code; and (3) we define the Map pattern and set parameter placeholders since the actual values change for each stream item, and compile the pattern. These steps are identified in Figure 4.4.

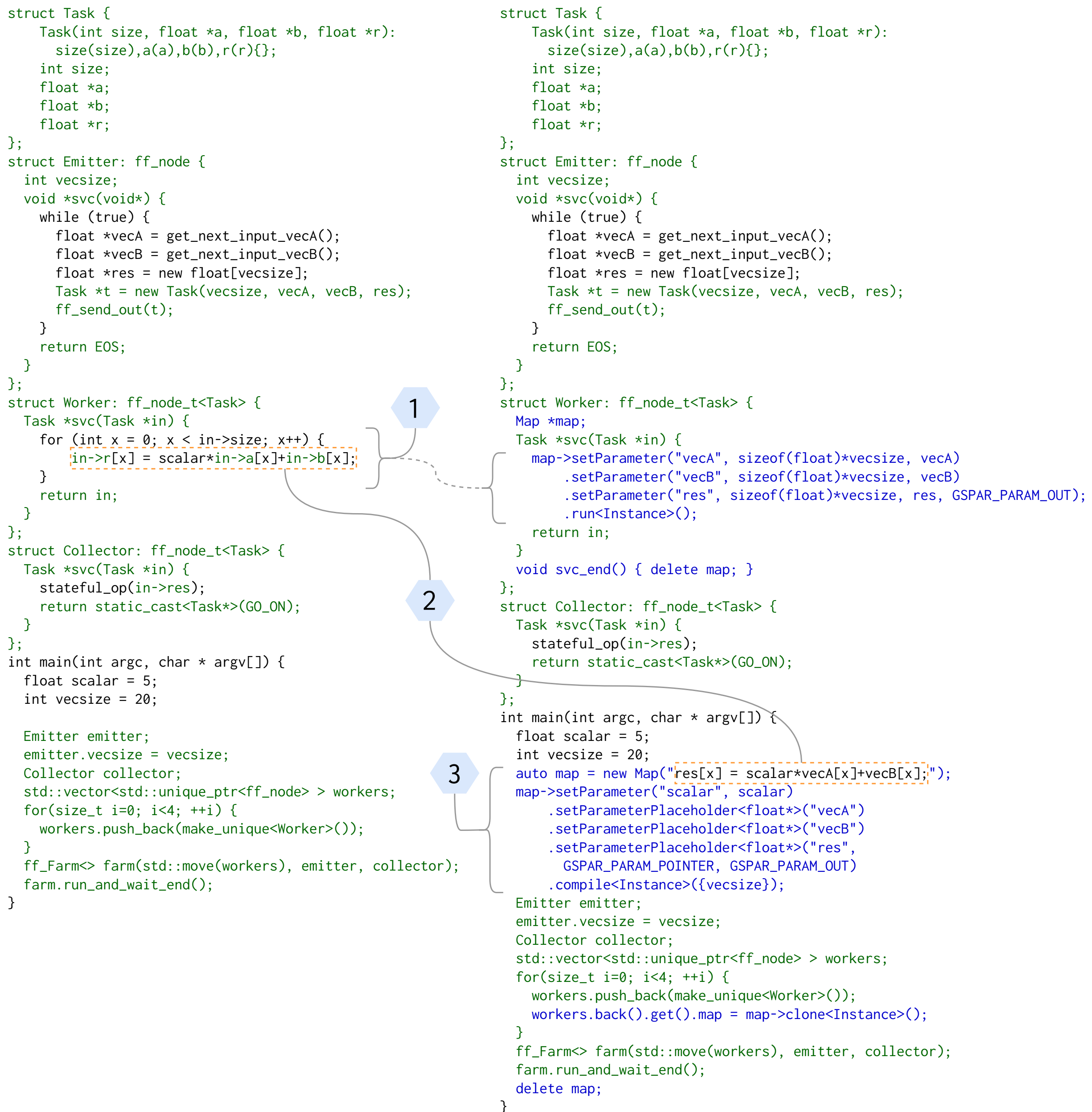


Figure 4.4: Applying GSPARLIB in FastFlow code.

Since the worker stage is executed in parallel, we need to create a clone of the pattern object for each parallel worker since it is not a thread-safe object. To this end, the pattern offers a `clone` method, which shares the thread-safe compiled pattern for all object clones and thus does not require recompilation when executing the pattern in parallel. This step is performed during the preparation of the workers, right before launching the Farm execution. Therefore, each parallel worker holds a clone of the Map pattern and can safely

invoke the pattern in parallel. These clones are thin objects with a light memory footprint, thus the application may create and hold many clones without any significant performance hit. GSPARLIB automatically manage the internal state of these objects and releases the underlying shared GPU kernel once all clones are destroyed.

Please note that none of the GSPARLIB code (highlighted in blue) in Figures 4.2 and 4.4 has any reference to CUDA nor OpenCL drivers. The underlying driver is defined based on the namespace of the `Instance` class passed as template argument to the methods `compile`, `clone`, and `run`. The `Instance` class is defined in two namespaces: `GSPar::Driver::CUDA` and `GSPar::Driver::OpenCL`. Therefore, to switch between the two drivers the programmer needs to change only the `using` clause to specify the namespace of the desired driver (this is not shown in the aforementioned Figures for the sake of simplicity). We recommend putting these `using` clauses inside a `ifdef` preprocessing directive to allow switching between the drivers using a compiler flag as will be demonstrated in Listing 4.2.

4.3 API and Runtime Design Choices

In this section we discuss some of the choices made during the GSPARLIB design. Firstly, we chose to separate the GSPARLIB API in two layers: a lower-level API, named Driver API, which acts as a wrapper over CUDA and OpenCL drivers; and a higher-level API, named Pattern API, focused in structured parallel programming. This layered design provide access to two abstraction levels for the same API, supporting system and application programmers according to their needs and expertise. The lower-level Driver API offers a driver-agnostic Object Oriented Programming (OOP) wrapper which transparently translate calls to CUDA and OpenCL drivers while the higher-level Pattern API offers support for Map and Reduce patterns as well as any composition of such patterns. This layered design is very similar with that of *Boost.Compute*, discussed in Section 3.1. Figure 4.5 shows the relationship between both layers and the structured parallel patterns. Future works may add support for more patterns using the same lower-level Driver API.

We also want to highlight our choice of exposing object oriented APIs. OOP is arguably one of the most used programming paradigms for software development worldwide since it is supported by most of the modern programming languages [TIO20]. Since the addition of classes is the single biggest difference between C and C++ [Str94], it makes little sense to develop a modern C++ library on any other programming paradigm. Unlike OpenCL, which offers C++ bindings in its current releases, latest CUDA versions does not include any C++ API for its Driver API. Thus, our library wrap these differences using the procedural APIs from both vendors and offering a single and unified object oriented layer [Bal08] with a fluent interface (with method chaining) [Fow10].

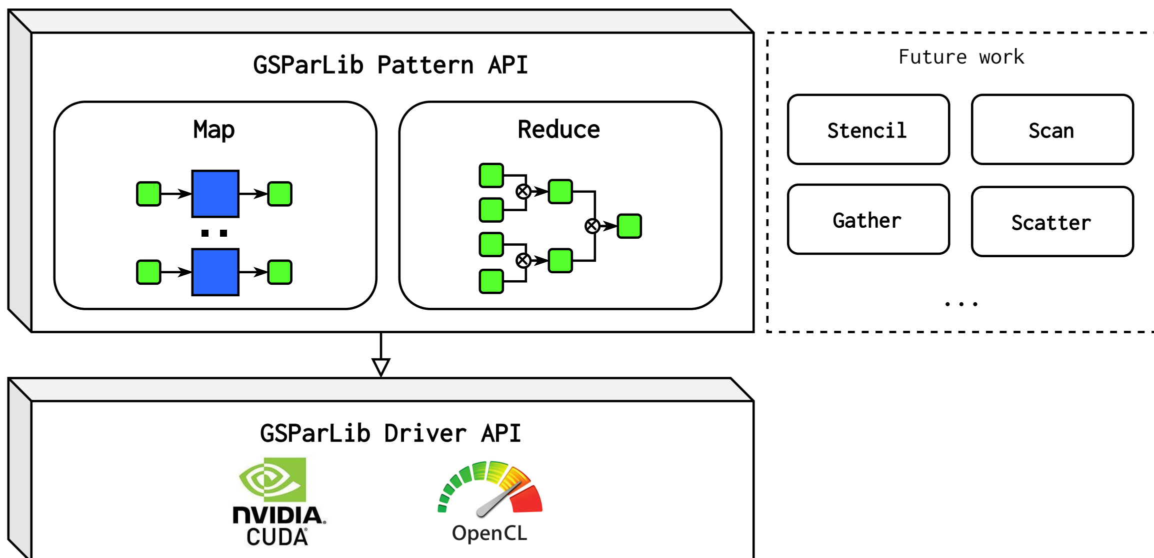


Figure 4.5: Overview of GSPARLIB APIs and parallel patterns.

We already discussed in Section 4.1 the decision of generating the GPU kernel code during runtime. As previously mentioned, this design allows to generate code optimized for the heterogeneous computer architecture which is running the application. Moreover, this decision contributes towards a unified API and a driver-agnostic runtime, aiming to abstract the differences of CUDA and OpenCL programming models. The CUDA Runtime API, which uses the `nvcc` compiler, provides generation and compilation of the GPU kernel code during compile-time. However, the OpenCL execution model always compiles the GPU kernel code during runtime. Using the CUDA Driver API and the NVIDIA runtime compiler (`nvrtc`), we can provide a unified approach for CUDA and OpenCL, which compiles the GPU kernel code at runtime.

During the library design we had to made a choice concerning page-locked host memory. The CUDA driver only allows for overlapping memory transfers and computations if the host memory is pinned (non-pageable) [Har12b]. This feature is specially important for stream processing applications since one pipeline stage may perform a memory copy while other stage is launching a GPU kernel. Therefore, GSPARLIB's `MemoryObject` offers a `pinHostMemory` method to page-lock the host memory, which is always called by the patterns from GSPARLIB's Pattern API right after the GPU memory is allocated. In its current version, GSPARLIB's `pinHostMemory` uses `cuMemHostRegister` in CUDA driver and does nothing in OpenCL driver, as the OpenCL specification does not require the use of page-locked memory to overlap memory copies and GPU kernel execution [Khr18]. Defining the macro `GSPAR_PATTERN_DISABLE_PINNED_MEMORY` disables this automatic memory pinning from the Pattern API.

Another design choice is to allow interoperability of the GSPARLIB Driver API with the lower-level CUDA and OpenCL procedural APIs by exposing methods to access the underlying CUDA/OpenCL objects which are wrapped in the GSPARLIB classes. OOP advo-

cates that these objects should be entirely abstracted and encapsulated by the class [Bal08]. However, in practice, many times the application programmer have to deal with challenges unanticipated by the system programmer. We design the abstraction layer of GSPARLIB to leak the abstraction [Spo02], allowing the application programmer to overcome these challenges. Otherwise, the application programmer would be unable to proceed using the library or would have to sacrifice performance to overcome it (e.g. we had to perform an extra memory copy to use SkePU 3 data containers due to differences between the container which was holding the data and the container the API was expecting. More on that in Section 4.7). This versatility also allows the programmer to apply specific optimizations, while they are not implemented in GSPARLIB yet, such as shared memory usage and coalesced memory access.

Now we discuss how GSPARLIB follows the principles for tools based on structured parallel programming discussed in Section 2.2 [Col04]:

1. **Propagate the concept with minimal conceptual disruption**. GSPARLIB is implemented in the main programming paradigm (OOP) of a widely used language (C++). The lower-level runtimes are the *de facto* standard CUDA and OpenCL. Each of the pattern classes are clearly defined as one parallel pattern, both in terms of naming and usability.

2. **Integrate ad-hoc parallelism.** GSPARLIB was built to be usable in multi-threaded environments. All classes are either thread-safe or offer a way to be used in multiple threads (such as cloning the object). Thus, GSPARLIB can be safely used together with other tools focused in parallel programming.

3. **Accommodate diversity.** GSPARLIB offers a way to combine multiple Map and Reduce pattern instances. Moreover, the function to be applied for each data element in the Map pattern is provided by the user, allowing arbitrary operations to be performed over the input data.

4. **Show the pay-back.** GSPARLIB offers clear advantages in various aspects of GPU programming, such as programmability (Section 4.6) and performance (Section 4.7). We also provide access to two abstraction levels for the same API, supporting system and application programmers according to their needs and expertise.

The main purpose of GSPARLIB library is to be integrated with another tool for multi-core parallelism and used in stream processing applications. This was a major driver in design choices such as: (a) define pattern and GPU kernel parameters as inputs, outputs, or both, which are the definitions commonly seen in this class of applications; (b) offer asynchronous operations whenever possible so that a pipeline stage can issue an asynchronous operation and send the task to the next stage, thus the operation is executed during the

communication between stages; (c) support thread-safety to permit the exploitation of heterogeneous parallelism, mainly to allow various pipeline stages replicated to execute patterns simultaneously.

4.4 GSPARLIB: low-level Driver API

The main motivation behind the Driver API is to act as a single interface to the different GPU drivers, currently CUDA and OpenCL. In addition to a unified interface, it is able to abstract from the GPU programmer the complexities of lower-level language constructs (such as C-like error handling) and provide a unified C++ programming API. This lower-level API is intended to be used by system programmers seeking a unified C++ object oriented API for GPU programming.

Table 4.1 links GPU programming concepts to the objects in the CUDA Driver API, OpenCL API, and GSPARLIB Driver API. Some GPU programming concepts are abstracted away entirely from the programmer in GSPARLIB (such as the compilation of the kernel and GPU context management) while others are wrapped in helper classes to ease C++ programming (such as asynchronous processes and exception handling). Nonetheless, most classes in GSPARLIB Driver API represent existing objects in CUDA or OpenCL APIs such as `Device`, `Kernel`, and `ExecutionFlow`.

Table 4.1: GSPARLIB Driver API analogous concepts.

GPU concept	*CUDA (Driver API)*	*OpenCL*	GSPARLIB *(Driver API)*
Kernel compilation	`nvrtcProgram` `CUmodule`	`cl_program`	Implicit in `Kernel`
Initialization	`cuInit()`	None	`Instance::init()`
Context	`CUcontext`	`cl_context`	Automatically managed by `Device`
Device	`CUdevice`	`cl_device_id`	`Device`
Memory allocation	`CUdeviceptr`	`cl_mem`	`MemoryObject` `ChunkedMemoryObject`
GPU Kernel	`CUfunction`	`cl_kernel`	`Kernel`
Commands channel	`CUstream`	`cl_command_queue`	`ExecutionFlow`
Asynchronous flow	`CUstream`	`cl_event`	`AsyncExecutionSupport`
Error handling	`CUresult` `nvrtcResult`	`cl_int`	`Exception`
Launch dimensions	`unsigned int` in `cuLaunchKernel`	`size_t[3]` in `clEnqueueNDRangeKernel`	`Dimensions struct` in `Kernel::runAsync`

There are four main classes in the Driver API, where each one represents an important entity of the domain of GPU programming. We present a summary of them below:

- `Device`: each object of this class represents a single GPU device. These objects are thread-safe so that they can be safely used simultaneously by many parallel threads. We

recommend that only a single Device object is used for each GPU in the heterogeneous system, however, we do not enforce because it may be easier for the programmer to create multiple Device objects to manage the same GPU;

- MemoryObject: each object of this class represents a block of memory allocated in a single Device. It can be instantiated by calling Device::malloc or by invoking the class constructor directly;

- Kernel: each object of this class represents a single GPU kernel. These objects can be instantiated by calling Device::prepareKernel or by invoking the class constructor directly. They are not thread-safe, but each parallel thread can make its own copy of the Kernel object by using the Kernel::cloneInto method;

- Instance: this class acts as a thread-safe entry point of the API. It is implemented using the *Singleton* design pattern [GHJV94], which means that the API enforces that only one instance of this class exists. This class instance can be accessed by using the Instance::getInstance method.

The Driver API class and methods have simple names, directly related to the domain of GPU programming, which should be recognized by any GPU programmer. The Instance class acts as a starting point for GPU programming using the API. GPU devices and kernels are represented by the Device and Kernel classes while blocks of memory allocated in some device are represented by the MemoryObject class. The Instance class offers methods for initialization of the GPU driver (init), querying the number of devices (getGpuCount) and getting Device objects (getGpu). Memory objects may be allocated by calling Device::malloc or by directly constructing a new MemoryObject.

Figure 4.6 presents a Unified Modeling Language (UML) diagram of all the relationships between the Driver API classes. There are two classes for error handling defined in the GSPar::Driver namespace: GSParException is used for general exceptions that are generated without direct relation to the underlying driver (CUDA or OpenCL); and Exception inherits the first and is thrown mostly due to errors reported by the underlying driver. This scheme permits the application to catch any of GSParLib's exceptions in a single catch clause referring to the GSParException class.

The Dimensions structure is used to define the lower and upper bounds for each dimension when executing the GPU kernel. It simply groups three references for the SingleDimension structure, one for each dimension. Even though the current GSParLib version only supports executing GPU kernels up to two dimensions, these structures were designed aiming at the standard three-dimensional execution grid of CUDA and OpenCL. The ExecutionFlow class acts as a wrapper for cl_command_queue and CUstream data types, as detailed in Table 4.1. Each device has its own default ExecutionFlow instance, but any number of flows may be created for each device. An operation enqueued in a ExecutionFlow

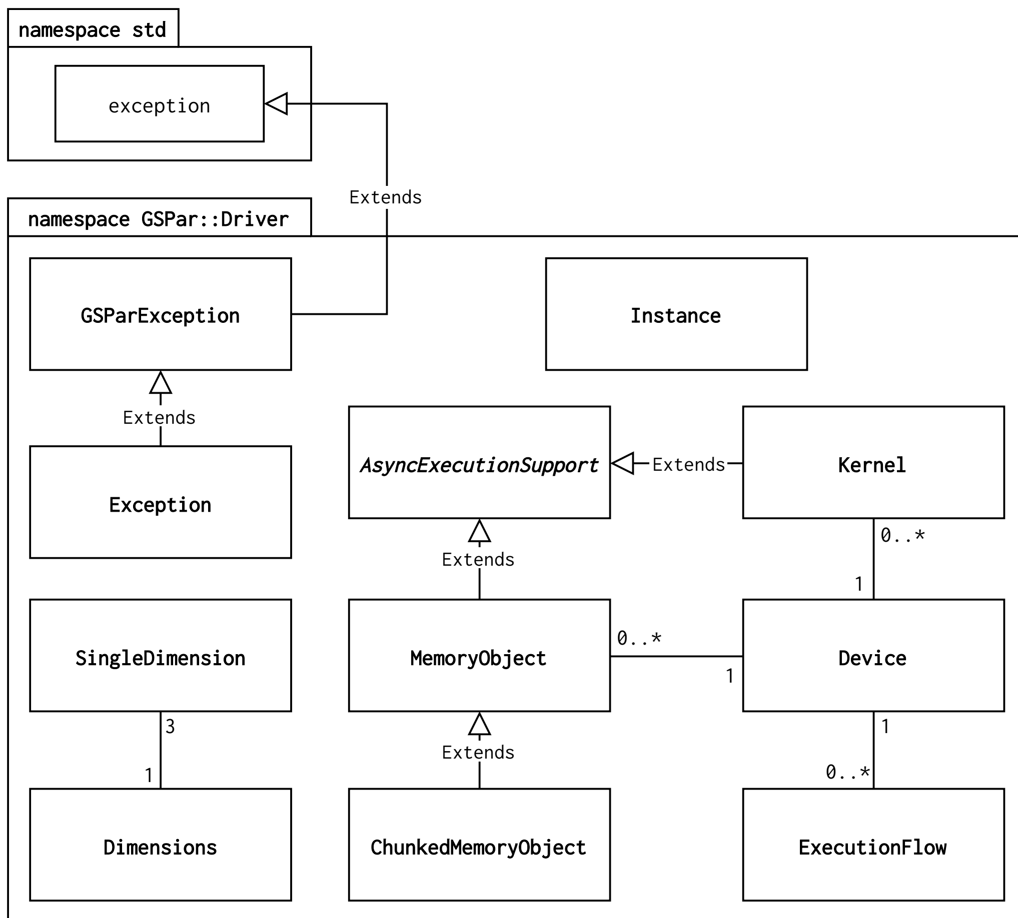


Figure 4.6: UML class relationships of the Driver API.

is guaranteed to be processed only after all the previous operations on this flow have been finished. Thus, the programmer may create different flows to overlap different operations, such as memory copies and GPU kernel execution. The `AsyncExecutionSupport` abstract class provides common functionality for querying whether any asynchronous operation is running and waiting this operation to finish execution. It is inherited by the `Kernel` and `MemoryObject` classes, which are the classes that provide a sort of asynchronous execution: running the GPU kernel and copying memory.

Programming an application using the Driver API follows the standard GPU programming flow defined by tools like CUDA and OpenCL, which can be summarized in 4 steps: (1) initialize the API and identify the devices; (2) allocate device memory and copy the necessary data; (3) prepare the `Kernel` object and invoke the GPU kernel; (4) copy results back to the main (host) memory and release resources. These steps can be seen in the sample vector sum application using the Driver API presented in Listing 4.1. This application simply sums two vectors (variables `a` and `b`) of the same size (defined by the `size` variable), and stores the sum in the variable `result`. Each vector element can be summed independently from the other elements, thus this program can be safely offloaded to exploit GPU parallelism.

We chose CUDA or OpenCL drivers at compile-time defining the macros `GSPARDRIVER_CUDA` or `GSPARDRIVER_OPENCL` by using the `-D` compiler flag. We define the underlying driver using preprocessing directives, according to these macros in lines 1–24 of Listing 4.1. In the lower-level Driver API, the GPU kernel source code must be provided as string (for which we use the `GSPAR_STRINGIZE_SOURCE` macro in lines 4 and 16) for each of the drivers individually. All Driver API classes are defined under the `GSPar::Driver` namespace, thus we employ the `using` clause in lines 3 and 15 to avoid having to write the full namespace name when using the API classes.

```
#ifdef GSPARDRIVER_CUDA
  #include "GSPar_CUDA.hpp"
  using namespace GSPar::Driver::CUDA;
  const char* kernelSource = GSPAR_STRINGIZE_SOURCE(
    extern "C" __global__
    void kernelVectorSum(const int size, const int *a, const int *b, int
   *result) {
      size_t x = blockIdx.x * blockDim.x + threadIdx.x;
      if (x <= size) {
        result[x] = a[x] + b[x];
      }
    }
  );
#else
  #include "GSPar_OpenCL.hpp"
  using namespace GSPar::Driver::OpenCL;
  const char* kernelSource = GSPAR_STRINGIZE_SOURCE(
    __kernel void kernelVectorSum(const int size, __global const int *a,
   __global const int *b, __global int *result) {
      size_t x = get_global_id(0);
      if (x <= size) {
        result[x] = a[x] + b[x];
      }
    }
  );
#endif

void vector_sum(const int size, const int *a, const int *b, int *result) {
  try {
    Instance* driver = Instance::getInstance();
    driver->init();

    auto gpu = driver->getGpu(0); // Get the first GPU

    MemoryObject* devA = gpu->malloc(sizeof(int) * size, a);
    MemoryObject* devB = gpu->malloc(sizeof(int) * size, b);
    devA->copyInAsync();
    devB->copyInAsync();
    AsyncExecutionSupport::waitAllAsync({ devA, devB });

    MemoryObject* devResult = gpu->malloc(sizeof(int) * size, result);

    // Kernel* kernel = gpu->prepareKernel(kernelSource, "kernelVectorSum");
    Kernel* kernel = new Kernel(gpu, kernelSource, "kernelVectorSum");

    kernel->setParameter(sizeof(size), &size);
    kernel->setParameter(devA);
```

```
46      kernel->setParameter(devB);
47      kernel->setParameter(devResult);
48
49      kernel->runAsync({size, 0});
50      kernel->waitAsync();
51
52      devResult->copyOut();
53
54      delete kernel;
55      delete devA; // Releases device memory
56      delete devB;
57      delete devResult;
58    } catch (GSPar::GSParException &ex) {
59      std::cerr << "Exception: " << ex.what() << " - " << ex.getDetails() <<
       std::endl;
60    }
61  }
```

Listing 4.1: Vector sum using the Driver API.

The `Instance` singleton is used to access a specific GPU (in Listing 4.1 we access the first GPU in line 31) or to list all the GPUs using the `getGpuList` method. To allocate memory in the GPU we use the `malloc` method of the `Device` class, optionally binding a host pointer to the allocated device memory. The `malloc` method returns a pointer to a single `MemoryObject` instance, which can be used to copy data between the GPU and the host memories. In lines 33, 34, and 39 of Listing 4.1 we allocate GPU memory for the vectors `a`, `b`, and `result` as well as store the resulting `MemoryObject` pointers in the variables `devA`, `devB`, and `devResult`, respectively. The synchronization between host and device memory is done explicitly by calling the copy methods of the `MemoryObject` objects: by calling `copyIn` or `copyInAsync` the memory is copied from the host to the device while `copyOut` or `copyOutAsync` methods can be used to copy the memory from the device to the host. In lines 35 and 36 of Listing 4.1 we use the `copyInAsync` method to asynchronously copy the input vectors `a` and `b` from the host to the GPU memory. When an asynchronous operation is started, the `waitAsync` method can be used on the object to wait for the operation to complete. This method is defined in the `AsyncExecutionSupport` class, which is inherited by classes that allows for asynchronous operations, such as `Kernel` and `MemoryObject`. Optionally, many asynchronous operations can be waited together by using the static method `AsyncExecutionSupport::waitAllAsync`, as demonstrated in line 37 of Listing 4.1.

The GPU kernel is prepared by calling `Device::prepareKernel` (line 41) or by using the `Kernel` class constructor (line 42). Any of these calls set-ups the kernel compilation by CUDA (using the NVIDIA runtime compilation, NVRTC) or OpenCL drivers, which may be a time-consuming operation. Since the `Kernel` objects are not tread-safe, this class provides the `cloneInto` method to make a clone of the `Kernel` object for each parallel thread of the program. The cloning operation ensures that all cloned objects use the same compiled kernel instead of compiling a different kernel for each thread.

The argument value for each parameter declared in the GPU kernel (lines 6 and 17 for CUDA and OpenCL, respectively) must be passed to the `Kernel` object in the same order as they were declared in the kernel source. These kernel arguments are set by calling the `setParameter` method and passing as arguments the size and pointer to the value (line 44) or the `MemoryObject` (lines 45–47).

After setting the parameters, the kernel is executed by calling `Kernel::runAsync` method in line 49, passing as argument a `Dimensions` two-dimensional structure that represents the last (exclusive) value of the index variable (also called end index or maximum value). The programmer may also define the first (inclusive) value of the variable (also called start index or minimum value) in the `Dimensions` structure, which has a default value of 0. The number of threads and blocks launched by GSParLib in the GPU is defined by the difference (delta) of the maximum and minimum values in this `Dimensions` structure as well as the GPU thread and block size limits. Deleting the memory objects (lines 55–57) frees the related device memory.

The Driver API also offers support for setting shared memory allocation for the kernel. This can be done by calling `Kernel::setSharedMemoryAllocation(unsigned int)`, where the argument is the size in bytes of the shared memory that should be allocated for each thread block. Since this memory is shared block-wide, the programmer may need to know the grid and block sizes (number of blocks and number of threads that will be launched) for a specific `Dimensions` structure. Thus, this information can be obtained by calling the `Kernel::getNumBlocksAndThreadsFor(Dimensions)` method.

The Driver API also provides a wrapper over CUDA and OpenCL error codes using modern C++ exception handling. The `GSParException` class serve as a base class for specific `Exception` classes of each driver (CUDA and OpenCL). They can all be caught in a single `catch` block, as demonstrated in line 58 of Listing 4.1. Besides driver-specific exceptions, the Driver API itself also throws exceptions related to unavailable GPUs and programming mistakes (such as trying to set a memory object as read-only and write-only simultaneously).

In addition to the `MemoryObject` class, the Driver API also offers a specialized class for linking multiple individual host pointers to a single block of device memory, the `ChunkedMemoryObject` class. Objects of this specialized class can be created by calling the `mallocChunked` method of the `Device` class or by the using the `ChunkedMemoryObject` class constructor. In this class, the methods for copying (synchronizing) data between the host and device memories (`copyIn`, `copyInAsync`, `copyOut`, and `copyOutAsync`) receive an extra argument to specify if it should copy all chunks of memory or a specific chunk. This class eases batching small memory transfers into a single transfer, which is a recommended approach to improve performance in GPU programming [Har12a]. However, the programmer is in charge of computing the indexes for accessing the memory object in the kernel, which can be a cumbersome task. Nonetheless, the main use of this class is to support the batch feature of the Pattern API (which will be discussed in Section 4.5).

GSPARLIB supports interoperability with the lower-level CUDA and OpenCL APIs by providing methods to access the underlying CUDA and OpenCL objects. The programmer can access the `CUstream` or `cl_command_queue` object of an `ExecutionFlow` using the `getBaseFlowObject` method. Similarly, the `CUstream` or `cl_event` that represents an asynchronous operation in `MemoryObject` (copying memory) or `Kernel` (executing the GPU kernel) can be accessed using the `getBaseAsyncObject` method. Other lower-level objects exposed through the Driver API are the objects that represent a device (`CUdevice` and `cl_device_id`), which can be accessed using the `Device::getBaseDeviceObject` method, and the objects that represents a memory allocation in the GPU (`CUdeviceptr` and `cl_mem`), which can be accessed by using the `getBaseMemoryObject` method from the `MemoryObject` class.

The Driver API also offers auxiliary functions to abstract some of the differences between CUDA and OpenCL kernel programming, which are meant to be called inside the GPU kernel code:

- `gspar_get_global_id(dim)`: returns the global ID of the current thread for the dimension (0, 1, or 2, which represents x, y, and z dimensions, respectively). The OpenCL implementation of this function calls `get_global_id(dim)` and the CUDA implementation calculates the global ID using `blockIdx.D * blockDim.D + threadIdx.D`, where `D` represents the dimension name (x, y, or z);
- `gspar_get_thread_id(dim)`: returns the local ID of the current thread inside the block. It is represented by OpenCL's `get_local_id(dim)` and CUDA's `threadIdx.D`;
- `gspar_get_block_id(dim)`: returns the the current thread block's ID, which is represented by the `get_group_id` function in OpenCL and the `blockIdx` structure in CUDA;
- `gspar_get_block_size(dim)`: returns the the current thread block's size, which is represented by the `get_local_size` function in OpenCL and the `blockDim` structure in CUDA;
- `gspar_synchronize_local_threads()`: performs a synchronization barrier for the threads of the same block or work group. This function calls `__syncthreads()` in CUDA and `barrier(CLK_LOCAL_MEM_FENCE)` in OpenCL.

4.5 GSPARLIB: high-level Pattern API

The Pattern API provides a higher-level structured programming interface that makes use of the Driver API capabilities to support CUDA and OpenCL alike. Currently, the Pattern API supports the Map and Reduce parallel patterns by providing homonymous classes. These patterns may also be bundled together in a pattern composition.

Figure 4.7 presents a UML diagram of the Pattern API classes relationships. All classes of this API are defined inside the `GSPar::Pattern` namespace. The `BaseParallelPattern` is an abstract class that provides general functionality for all parallel patterns. Thus, most of the API behavior is defined in this class, which is inherited by the `Map` and `Reduce` classes. These pattern classes overload some of the methods of the main class to implement pattern-specific behavior. The `PatternComposition` class represent a collection of patterns that should be compiled and executed together. Currently, the `PatternComposition` class always executes the patterns sequentially, but support for custom pattern dependencies or different schedulers may be implemented by future works.

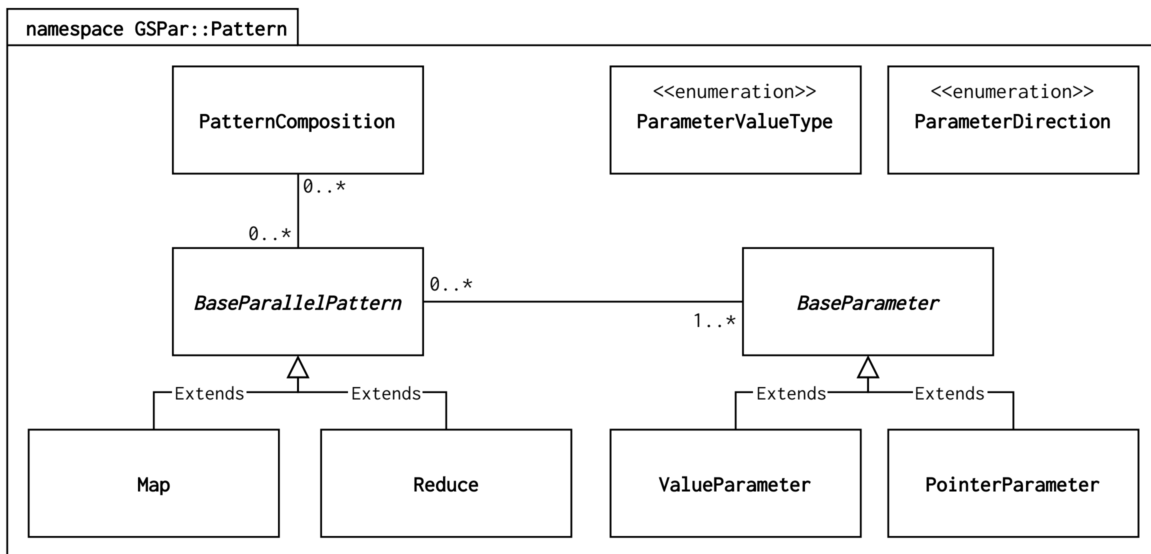


Figure 4.7: UML class relationship of the Pattern API.

In addition to those main classes, there are two enumerations: `ParameterValueType` is used to differentiate parameters defined as built-in data types (value) and pointers; and `ParameterDirection` is used to identify in which directions the parameter should be copied with relation to the GPU memory, in, out, in-out, and none (the parameter is not copied). Each of the pattern's parameters is a specialized instance of the `BaseParameter` class, which have only two specializations: `ValueParameter` for parameters of built-in data types and `PointerParameter` for parameters defined as pointers.

4.5.1 Map Pattern

The Map pattern offers a simple interface that allows programmers to execute multiple iterations of the same operation in parallel. Listing 4.2 demonstrates the same vector sum program from Listing 4.1, now using the higher-level Pattern API instead of the lower-level Driver API. As a matter of fact, Listing 4.2 presents the complete code of the same program depicted in Figure 4.2. As a lightweight library, GSParLib could not inspect the

source code to automatically generate the kernel based on lambda or user-defined functions in the C++ source code, such as SkePU and PACXX. Therefore, the core operation of the GPU kernel must still be passed as string, for which we use the `GSPAR_STRINGIZE_SOURCE` macro in line 13. Nevertheless, differently from the Driver API kernel, which requires the user to pass the full kernel code (Listing 4.1), the `Map` class automatically generates the boilerplate platform-dependent code. The `Map` class also automatically defines standard variables called `x` and `y` for the global thread index in the two dimensions of the GPU kernels, using the `gspar_get_global_id` function. These names can be customized using the `setStdVarNames` method.

```
#ifdef GSPARDRIVER_CUDA
  #include "GSPar_CUDA.hpp"
  using namespace GSPar::Driver::CUDA;
#else
  #include "GSPar_OpenCL.hpp"
  using namespace GSPar::Driver::OpenCL;
#endif
#include "GSPar_PatternMap.hpp"
using namespace GSPar::Pattern;

void vector_sum(const int size, const int *a, const int *b, int *result) {
  try {
    auto pattern = new Map(GSPAR_STRINGIZE_SOURCE(
      result[x] = a[x] + b[x];
    ));
    pattern->setParameter("size", size)
      .setParameter("a", sizeof(int) * size, a)
      .setParameter("b", sizeof(int) * size, b)
      .setParameter("result", sizeof(int) * size, result, GSPAR_PARAM_OUT)
      .run<Instance>({size, 0});
    delete pattern;
  } catch (GSPar::GSParException &ex) {
    std::cerr << "Exception: " << ex.what() << " - " << ex.getDetails() <<
   std::endl;
  }
}
```

Listing 4.2: Vector sum using the Map pattern from the Pattern API.

After creating the pattern with the core kernel code in lines 13–15 of Listing 4.2, the pattern parameters are defined using the `setParameter` method (lines 16–19). This method has various overloads for different types of parameters and takes as arguments: a string containing the parameter name inside the kernel; the size of the parameter (only if the third argument is a pointer); the pointer to the values or the value if the parameter is a single value of one of C++ basic data types (such as the `size` parameter in line 16); and a flag specifying if the argument should be copied from the GPU memory to the host memory after computing the Map pattern (by default the parameter is only copied into the GPU memory).

After setting the parameters, the pattern is synchronously launched by calling the `run` method (line 20), which takes a reference to the `Instance` class of the Driver API as a template argument. This reference defines what driver (CUDA or OpenCL) will be used to compile

and run the pattern. The argument of the run method is the same as the Kernel::runAsync method from the Driver API, which is the Dimensions structure that defines the number of parallel threads that will be launched. By default, the first GPU is used to compile and run the kernel, however, it is possible to set a specific device for the pattern by using the setGpuIndex or setGpu methods.

The GPU kernel is automatically compiled whenever the run method is called. Alternatively, the pattern's GPU kernel can be compiled by calling the compile method with the Dimensions structure before the run method is called. In this case, the Dimensions may be omitted from the run call, as the pattern uses the configuration which was set during the compilation. The compiled GPU kernel is stored and reused in subsequently calls to the run method with the same Dimensions. The pattern objects are not thread-safe since they reference a Kernel object from the Driver API. Thus, they provide the clone method to create a copy of the pattern which also clones the compiled kernel and avoids the need to recompile it for each parallel execution.

GSPARLIB also supports compiling the pattern without the actual argument values if they are not available yet. The method setParameterPlaceholder is used to define placeholders for the arguments (lines 17–19 in Listing 4.3), passing the type as template argument. Besides this argument, this method also receives: the name of the parameter which is used inside the kernel; the type (pointer or single value); a flag specifying in which direction the argument should be copied between the GPU and host memories (possible values are IN, OUT, or INOUT); and a boolean flag specifying if the parameter is a batched value.

Listing 4.3 presents a summation of multiple independent vectors that would usually require a separate kernel launch for each operation. Instead, GSPARLIB receives a batch of vectors and allows a single kernel launch to operate in separate vectors simultaneously. The sum_vectors function receives as argument the number of vector pairs to be summed (numVectors), how many vector sum operations should be performed in each kernel call (batchSize), the size of each vector (vectorSize), and the pointers for the input and output vectors (as, bs, and results). The Map pattern is declared in lines 13–15 with the kernel core for the vector sum operation. Since the vector size is constant for all vectors, it is set as a pattern argument in line 16 while the rest of the parameters are set as placeholders (lines 17–19) because they are different for each batch that has to be computed. We define the batch size as an integer value that is the number of GPU kernel launches that should be combined, which is set in line 20. In Listing 4.3, the batch size represents the number of vector summations that should occur in a single run call. After defining the batch size, we compile the GPU kernel in line 21. This step is optional since the run method automatically compiles the GPU kernel if it was not compiled yet. Each run call will execute a single batch, thus we need to calculate the number of batches in line 22 that is the number of run calls to be issued. Nonetheless, before running the GPU kernel we need to replace the placeholder parameters by the actual argument values. In Listing 4.3, all placeholder parameters are replaced by

batched parameters with calls to `setBatchedParameter` in lines 27–29. The arguments of `setBatchedParameter` refer to a single element of the batch, thus it is very similar to the `setParameter` method. However, `setBatchedParameter` always receive pointers (or pointers of pointers) with `batchSize` elements.

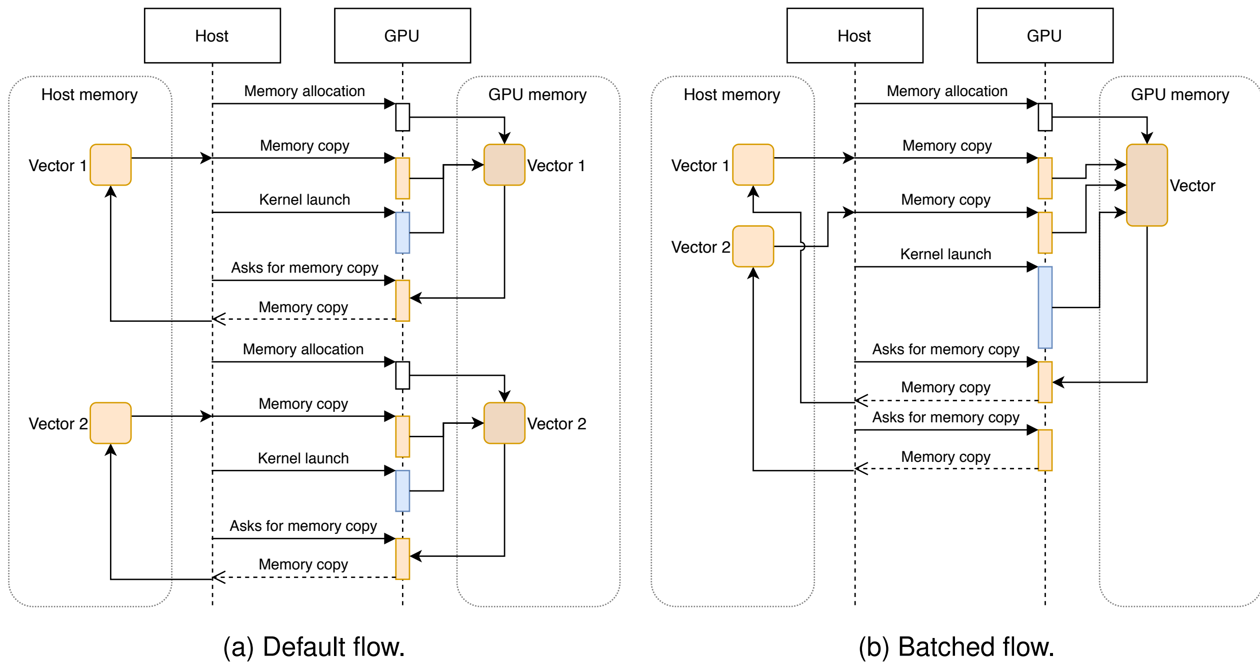
```
#ifdef GSPARDRIVER_CUDA
  #include "GSPar_CUDA.hpp"
  using namespace GSPar::Driver::CUDA;
#else
  #include "GSPar_OpenCL.hpp"
  using namespace GSPar::Driver::OpenCL;
#endif
#include "GSPar_PatternMap.hpp"
using namespace GSPar::Pattern;

void sum_vectors(int numVectors, int batchSize, int vectorSize, int **as, int
    **bs, int **results) {
  try {
    auto pattern = new Map(GSPAR_STRINGIZE_SOURCE(
      result[x] = a[x] + b[x];
    ));
    pattern->setParameter("size", vectorSize)
      .setParameterPlaceholder<int*>("a", GSPAR_PARAM_POINTER, GSPAR_PARAM_IN,
   true)
      .setParameterPlaceholder<int*>("b", GSPAR_PARAM_POINTER, GSPAR_PARAM_IN,
   true)
      .setParameterPlaceholder<int*>("result", GSPAR_PARAM_POINTER,
   GSPAR_PARAM_OUT, true)
      .setBatchSize(batchSize)
      .compile<Instance>({vectorSize, 0});
    unsigned int batches = numVectors/batchSize;
    for (unsigned int b = 0; b < batches; b++) {
      int *vecA = &as[b*batchSize];
      int *vecB = &bs[b*batchSize];
      int *vecRes = &results[b*batchSize];
      pattern->setBatchedParameter("a", sizeof(int)*vectorSize, vecA)
        .setBatchedParameter("b", sizeof(int)*vectorSize, vecB)
        .setBatchedParameter("result", sizeof(int)*vectorSize, vecRes,
   GSPAR_PARAM_OUT)
        .run<Instance>();
    }
    delete pattern;
  } catch (GSPar::GSParException &ex) {
    std::cerr << "Exception: " << ex.what() << " - " << ex.getDetails() <<
   std::endl;
  }
}
```

Listing 4.3: Batched vector sum using the Map pattern.

Figure 4.8 illustrates the differences between the default execution flow and the batched flow for an operation to be applied on two independent vectors. The default (sequential) flow (Figure 4.8a) is presented in Listing 4.2. In this flow, when a single vector is processed at a time: the host allocates memory for the vector in the GPU memory, copies data into the GPU, launches the GPU kernel, and asks for the GPU to copy the data back to host memory. Then it perform the same steps for the next vectors. The batched flow

(Figure 4.8b illustrates a batch size of 2) is presented in Listing 4.3. A single block of memory is allocated in the GPU to hold the data of the entire batch of vectors. Then the vectors are copied to the GPU memory and the kernel is launched. GSPARLIB automatically calculates the memory size to hold the data of the entire batch and the amount of threads needed to perform both computations in parallel. After the computation, the batch of vectors is copied back to host memory.



(a) Default flow. (b) Batched flow.

Figure 4.8: Default vs batched flow of execution.

The batching optimization reduces the number of GPU kernel invocations by merging multiple kernel invocations into a single one. The number of GPU kernel invocations to be merged is defined by the batch size. This increases the amount of work performed by the merged invocation and reduces the CPU-GPU communication. However, the result of any task in the batch become available only after the computation of the entire batch of tasks. Thus, for stream processing applications this optimization increases the latency of each task, but it also increases the overall throughput and reduces the total execution time [RSG+19, SRG+20]. We will discuss the performance impacts of this batching feature in Section 5.6.

4.5.2 Reduce Pattern

The Reduce pattern apply a function to multiple input data elements to combine them pairwise into a single output element. This specific computational pattern offers more opportunities for optimizations and an improved abstraction layer. By definition, the reductions can only be computed in parallel if the combiner function is associative, however, we chose to

support only associative and commutative operators because this permits many optimizations that provide interesting performance improvements, such as properly using data locality in the shared memory of GPU devices. Therefore, the `Reduce` class constructor takes three simple string arguments: the name of the parameter that will contain the vector of values to be reduced; the binary associative and commutative operator; and the name of the parameter that will contain the output of the Reduce operation. Listing 4.4 exemplifies these parameters in line 14. After declaring the pattern, the input and output parameters (`in_vector` and `total`, respectively) are defined in lines 15 and 16, using the same names used in the pattern constructor. Finally, the pattern is synchronously executed in line 17 by calling the `run` method.

```
1  #ifdef GSPARDRIVER_CUDA
2    #include "GSPar_CUDA.hpp"
3    using namespace GSPar::Driver::CUDA;
4  #else
5    #include "GSPar_OpenCL.hpp"
6    using namespace GSPar::Driver::OpenCL;
7  #endif
8  #include "GSPar_PatternReduce.hpp"
9  using namespace GSPar::Pattern;
10
11 int reduce_sum(const int size, const int *vector) {
12   int total;
13   try {
14     auto pattern = new Reduce("in_vector", "+", "total");
15     pattern->setParameter("in_vector", sizeof(int) * size, vector)
16         .setParameter("total", sizeof(int), &total, GSPAR_PARAM_OUT)
17         .run<Instance>({size});
18     delete pattern;
19   } catch (GSPar::GSParException &ex) {
20     std::cerr << "Exception: " << ex.what() << " - " << ex.getDetails() <<
      std::endl;
21   }
22   return total;
23 }
```

Listing 4.4: Summing a vector of values using the Reduce pattern.

The `Reduce` class automatically generates the GPU kernel using shared memory to perform the reductions, according to state-of-the-art optimizations [Har07]. Listing 4.5 present the CUDA kernel generated for Listing 4.4. The GPU kernel's first parameter (`max_x`) is the max value passed as argument to the `run` method. The second parameter (`in_vector`) is the input vector. The third parameter (`partial_reductions`) is a vector of partial reductions, which is used by GSParLib to accumulate partial reductions until the last reduction is performed. In OpenCL, there is a fourth parameter which is the shared memory. In CUDA the shared memory is defined as `gspar_shared` in the first line of the GPU kernel function body. Here, we use the GSParLib functions for common tasks to abstract programming differences between CUDA and OpenCL (discussed in Section 4.4), such as getting current thread and block IDs and sizes (lines 6–8 of Listing 4.5), as well as to synchronize threads in the same block (lines 10, 15, and 19).

```
extern "C" __global__
void gspar_reduce_kernel(const unsigned long max_x,int const* in_vector,int*
    partial_reductions) {
  extern __shared__ int gspar_shared[];
  size_t x = gspar_get_global_id(0);
  if (x < max_x) {
    size_t gspar_tid_x = gspar_get_thread_id(0);
    size_t gspar_bid_x = gspar_get_block_id(0);
    size_t gspar_bsize_x = gspar_get_block_size(0);
    gspar_shared[gspar_tid_x] = in_vector[x];
    gspar_synchronize_local_threads();
    for (unsigned int s=gspar_bsize_x/2; s>0; s>>=1) {
      if (gspar_tid_x < s && x+s < max_x) {
        gspar_shared[gspar_tid_x] += gspar_shared[gspar_tid_x+s];
      }
      gspar_synchronize_local_threads();
      if (gspar_tid_x == 0 && s > 1 && s % 2 != 0) {
        gspar_shared[gspar_tid_x] += gspar_shared[s-1];
      }
      gspar_synchronize_local_threads();
    }
    if (gspar_tid_x == 0) {
      if (gspar_bsize_x % 2 != 0) {
        gspar_shared[0] += gspar_shared[max_x-1];
      }
      partial_reductions[gspar_bid_x] = gspar_shared[0];
    }
  }
}
```

Listing 4.5: Generated Reduce GPU kernel for CUDA.

After obtaining their own position in the execution grid, each thread copies an element from the global memory to the shared memory (line 9 of Listing 4.5). Then, the reduction of all the elements in the same thread block is performed using strided index and a reversed loop (lines 11–20) to provide sequential memory addressing [Har07]. In line 22, the first thread of each block checks if there is an odd number of elements and includes this last element in the reduction. The resulting element of each thread block is then copied to the global memory in line 25. The automatic parallelization of the reduction operation performed by GSPARLIB poses additional challenges in the optimizations that can be applied. More aggressive optimizations such as loop unrolling and performing reductions during shared memory load, are very challenging to implement in a general-purpose library that must work correctly on any problem size.

Since each GPU thread block has its own shared memory, it is necessary to synchronize partial reductions across multiple thread blocks. However, there is no cross-block (global) synchronization command in CUDA nor in OpenCL [CGM14]. Therefore, each GPU kernel launched returns *N* results, where *N* is the number of thread blocks that were launched. After computing the block-wise reductions, we launch the same GPU kernel again using the partial reduction results as the input until the reduction is finished. The GPU kernel launch serves as a global synchronization point [Har07].

4.5.3 Pattern Composition

To combine the Map and Reduce patterns, GSPARLIB offers the `PatternComposition` class. Currently, the only supported combination is the sequential execution of the patterns. Listing 4.6 presents the combination of the Map and Reduce patterns presented in Listings 4.2 and 4.4, respectively, to sum all the elements of two vectors into a single element. After defining the Map (lines 14–19) and Reduce (lines 21–23) patterns, they are bundled together in a composition in line 25.

```
#ifdef GSPARDRIVER_CUDA
  #include "GSPar_CUDA.hpp"
  using namespace GSPar::Driver::CUDA;
#else
  #include "GSPar_OpenCL.hpp"
  using namespace GSPar::Driver::OpenCL;
#endif
#include "GSPar_PatternComposition.hpp"
using namespace GSPar::Pattern;

int vector_sum(const int size, const int *a, const int *b, int *result) {
  int total;
  try {
    auto map = new Map(GSPAR_STRINGIZE_SOURCE(
      result[x] = a[x] + b[x];
    ));
    map->setParameter("a", sizeof(int) * size, a)
      .setParameter("b", sizeof(int) * size, b)
      .setParameter("result", sizeof(int) * size, result, GSPAR_PARAM_OUT);

    auto reduce = new Reduce("result", "+", "total");
    reduce->setParameter("result", sizeof(int) * size, result,
   GSPAR_PARAM_INOUT)
        .setParameter("total", sizeof(int), &total, GSPAR_PARAM_OUT);

    auto mapReduce = new PatternComposition(map, reduce);
    mapReduce->run<Instance>({size, 0});

    delete mapReduce;
    delete reduce;
    delete map;
  } catch (GSPar::GSParException &ex) {
    std::cerr << "Exception: " << ex.what() << " - " << ex.getDetails() <<
   std::endl;
  }
}
```

Listing 4.6: Vector sum using pattern composition of Map and Reduce.

The `PatternComposition` class offers the `run` method (line 26 of Listing 4.6), which has the same signature as the homonymous method in the `Map` and `Reduce` classes. There is also the `compilePatterns` method, analogous to the `compile` method of the patterns' classes, which combines the sources of all the patterns in the composition and compiles them all

at once using the GPU driver. Patterns can be added in the composition using the class constructor (line 25 of Listing 4.6) or by using the `addPattern` method.

4.6 Programmability Considerations

In this section we will compare the programmability of GSPARLIB with some of the state-of-the-art tools. Firstly, we want to show an example of programming using Map and Reduce patterns in lower-level procedural APIs. Listing 4.7 shows the vector sum application in CUDA. We used the CUDA runtime API, which is less verbose and more commonly used than the driver API. However, it requires the use of the `nvcc` compiler.

```
#define CudaSafeCall( err ) __cudaSafeCall( err, __FILE__, __LINE__ )
inline void __cudaSafeCall( cudaError err, const char *file, const int line ) {
  if ( cudaSuccess != err ) {
    std::cerr << "CudaSafeCall() failed at " << file << ":" << line << " : " <<
   err << "-" << cudaGetErrorString(err) << std::endl;
    exit(-1);
  }
}
__global__ void vector_sum_map(int size, const int *a, const int *b, int *res) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if (gid < size) res[gid] = a[gid] + b[gid];
}
__global__ void vector_sum_reduce(int size, const int *res, int* total) {
  extern __shared__ int shmem[];
  int tid = threadIdx.x;
  int bid = blockIdx.x;
  int gid = bid * blockDim.x + tid;
  if (gid < size) {
    shmem[tid] = res[gid];
    __syncthreads();
    for (int s=blockDim.x/2; s>0; s>>=1) {
      if (tid < s && gid+s < size) shmem[tid] += shmem[tid+s];
      __syncthreads();
      if (tid == 0 && s > 1 && s % 2 != 0) shmem[tid] += shmem[s-1];
      __syncthreads();
    }
    if (tid == 0) {
      if (blockDim.x % 2 != 0) shmem[0] += shmem[size-1];
      total[bid] = shmem[0];
    }
  }
}
int vector_sum(const int vectorSize, const int *a, const int *b, int *result) {
  int totalDevices;
  CudaSafeCall(cudaGetDeviceCount(&totalDevices));
  if (totalDevices < 1) {
    std::cerr << "No CUDA-enabled device found" << std::endl;
    exit(-1);
  }
  cudaDeviceProp devProp;
  cudaGetDeviceProperties(&devProp, 0); // 0 is the first device
  // Prepares the map kernel
```

```
cudaFuncAttributes mapKernelAttrs;
CudaSafeCall(cudaFuncGetAttributes(&mapKernelAttrs, vector_sum_map));
unsigned int mapMaxThreadsPerBlock =
  sqrt(((double)devProp.regsPerBlock)/mapKernelAttrs.numRegs);
mapMaxThreadsPerBlock = std::min(mapMaxThreadsPerBlock,
  devProp.maxThreadsPerBlock);
unsigned int mapGridDim = 1;
unsigned int mapBlockDim = vectorSize;
if (mapBlockDim > mapMaxThreadsPerBlock) {
  mapGridDim = ceil((double)mapBlockDim / mapMaxThreadsPerBlock);
  mapBlockDim = mapMaxThreadsPerBlock;
}

// Prepares the reduce kernel
cudaFuncAttributes reduceKernelAttrs;
CudaSafeCall(cudaFuncGetAttributes(&reduceKernelAttrs, vector_sum_reduce));
unsigned int reduceMaxThreadsPerBlock =
  (((double)devProp.regsPerBlock)/reduceKernelAttrs.numRegs);
reduceMaxThreadsPerBlock = std::min(reduceMaxThreadsPerBlock,
  devProp.maxThreadsPerBlock);
unsigned int reduceGridDim = 1;
unsigned int reduceBlockDim = vectorSize;
if (reduceBlockDim > reduceMaxThreadsPerBlock) {
  reduceGridDim = ceil((double)reduceBlockDim / reduceMaxThreadsPerBlock);
  reduceBlockDim = reduceMaxThreadsPerBlock;
}

cudaStream_t stream1;
CudaSafeCall(cudaStreamCreate(&stream1));
cudaStream_t stream2;
CudaSafeCall(cudaStreamCreate(&stream2));

// To allow overlapping copies
CudaSafeCall(cudaHostRegister(a, vectorSize*sizeof(int),
  cudaHostRegisterDefault));
CudaSafeCall(cudaHostRegister(b, vectorSize*sizeof(int),
  cudaHostRegisterDefault));

int *aDev;
CudaSafeCall(cudaMalloc((void**)&aDev, vectorSize*sizeof(int)));
int *bDev;
CudaSafeCall(cudaMalloc((void**)&bDev, vectorSize*sizeof(int)));
int *resultDev;
CudaSafeCall(cudaMalloc((void**)&resultDev, vectorSize*sizeof(int)));
int *partialReductions;
CudaSafeCall(cudaMalloc((void**)&partialReductions, reduceGridDim *
  sizeof(int)));
CudaSafeCall(cudaMemcpyAsync(aDev, a, vectorSize*sizeof(int),
  cudaMemcpyHostToDevice, stream1));
CudaSafeCall(cudaMemcpyAsync(bDev, b, vectorSize*sizeof(int),
  cudaMemcpyHostToDevice, stream2));
CudaSafeCall(cudaStreamSynchronize(stream1));
CudaSafeCall(cudaStreamSynchronize(stream2));

vector_sum_map<<<mapGridDim,mapBlockDim,0,stream1>>>(vectorSize, aDev, bDev,
  resultDev);
CudaSafeCall(cudaStreamSynchronize(stream1)); // Waits the kernel to finish

// Launches the memory copy in the first stream
```

```
91    CudaSafeCall(cudaMemcpyAsync(result, resultDev, vectorSize*sizeof(int),
        cudaMemcpyDeviceToHost, stream1));
92
93    // For reduce we use the stream2 to overlap the memory copy above with the
        reduce computation
94    int *reduceInput = resultDev;
95    int reduceVectorSize = vectorSize;
96    while (true) {
97      size_t sharedMemSize = (reduceVectorSize > reduceBlockDim) ? reduceBlockDim
        : reduceVectorSize;
98      unsigned long sharedMemBytes = sizeof(int) * sharedMemSize;
99
100
        vector_sum_reduce<<<reduceGridDim,reduceBlockDim,sharedMemBytes,stream2>>>(reduceVector
        reduceInput, partialReductions);
101
102     if (reduceGridDim == 1) break; // Last reduction
103     // Calculate the next kernel launch sizes
104     reduceVectorSize = reduceGridDim;
105     reduceBlockDim = reduceGridDim;
106     if (reduceBlockDim > reduceMaxThreadsPerBlock) {
107       reduceGridDim = ceil((double)reduceBlockDim / reduceMaxThreadsPerBlock);
108       reduceBlockDim = reduceMaxThreadsPerBlock;
109     } else {
110       reduceGridDim = 1;
111     }
112     CudaSafeCall(cudaStreamSynchronize(stream2));
113     reduceInput = partialReductions;
114   }
115
116   int total;
117   CudaSafeCall(cudaMemcpy(&total, partialReductions, sizeof(int),
        cudaMemcpyDeviceToHost));
118
119   CudaSafeCall(cudaStreamSynchronize(stream1)); // Waits the memory copy of the
        map result to finish
120   // Release resources
121   CudaSafeCall(cudaFree(aDev));
122   CudaSafeCall(cudaFree(bDev));
123   CudaSafeCall(cudaFree(resultDev));
124   CudaSafeCall(cudaFree(partialReductions));
125
126   return total;
127 }
```

Listing 4.7: Vector sum using Map and Reduce patterns in CUDA runtime API.

We handle CUDA errors by using a macro and a function, respectively defined in the lines 1 and 2 of Listing 4.7. The Map kernel (lines 8–11) simply checks for extra threads and sums the two vectors. The Reduce kernel (lines 12–31) is very similar to those generated by GSParLib and discussed in Listing 4.5.

The first part of the main `vector_sum` function (line 32) discovers the resources and properties of the intended execution device (in this case, the first device) and calculates the block and grid launch sizes considering the device's maximum threads per block and maximum registers per block (lines 46–51 and 58–63). Then, two streams are created to

overlap multiple copy and kernel operations (lines 65–68). To allow overlapping memory copies and kernel launches, the host memory pointers are registered as page-locked memory (lines 70–72). After the memory is allocated and the data is copied into the GPU (lines 74–85), the Map kernel is launched (line 87).

We start copying the vector resulting of the Map pattern execution in line 91, which continues to copy while we start computing the Reduce pattern. The reduction kernel using shared memory was already described in Section 4.5.2. On the host side, we start a loop in line 96 which iterates until the reduction is finished. For each iteration, we calculate the amount of shared memory per block needed to perform the reductions (line 97) and launch the kernel (line 100). If there is more than one thread block, it means that we need to perform further reductions, so we recalculate the block and grid sizes (lines 104–111) and set the reduction result as the input for the next reduction iteration (line 113).

Once the reduce is finished, we copy the single output value to the host memory (line 116) and release the GPU memory allocations (lines 121–124). For the sake of simplicity we did not release the GPU memory in the case of a failure in any CUDA function call, since we are interrupting the program with `exit`.

We would like to highlight some important drawbacks of the CUDA implementation compared to the GSPARLIB's version of this same application (Listing 4.6):

1. error handling is much more verbose and error-prone, as one could easily forget to wrap a function call in `CudaSafeCall` macro. We commit this mistake on purpose in line 40 of Listing 4.7 as an example;

2. launch sizes must be calculated considering device and kernel properties, which is a verbose and toilsome task. Besides the block size and register per block limits, there is also other limiting factors, such as the available amount of shared memory per block (we did not perform this check in Listing 4.7);

3. streams must be explicitly created and synchronized, which is an error-prone task. Programming applications using multiple streams requires the programmer to fully understand the entire execution flow and the details of memory/computation latencies, to properly choose which stream to use for each operation in order to truly overlap memory copies and computation;

4. memory must be managed explicitly (which involves allocation, copies to and from the device, and releasing). It also needs to be explicitly registered as page-locked memory to allow overlapping operations;

5. resources must be explicitly released, even in the case of failures (Listing 4.7 does not release resources when a failure occurs);

6. if the program was to be executed in other device from a different vendor, it would need to be totally rewritten in OpenCL.

GSPARLIB provides effective advantages in all these aspects: (1) error handling is done using modern C++ exceptions, which are far easier to use and implement than the status code returned by procedural APIs [Str94]; (2, 3, and 4) launch sizes, streams, execution flows, and memory management are completely abstracted in GSPARLIB Pattern API; (5) resources are automatically released using the class destructor when the object pointer is deleted or when the object goes out of scope; (6) GSPARLIB support CUDA and OpenCL alike and abstracts the differences between the two APIs.

Now, we turn our attention over higher-level APIs. Listing 4.8 presents a simple matrix multiplication example using *Boost.Compute*, which is based on OpenCL. Since two-dimensional kernels are not supported in the *Boost.Compute* higher-level API, we use the lower-level API of *Boost.Compute* for this example, which requires passing the entire OpenCL kernel source as string (lines 3–14).

```
#include <boost/compute.hpp>

const char* kernelSource =
"__kernel void matrix_multi(unsigned long size, __global const float *a,
   __global const float *b, __global float *result) { \n"
" size_t i = get_global_id(0); \n"
" size_t j = get_global_id(1); \n"
" if (i < size && j < size) { \n"
"   float sum = 0; \n"
"   for (unsigned long k = 0; k < size; k++) { \n"
"     sum += a[k * size + i] * b[j * size + k]; \n"
"   } \n"
"   result[j * size + i] = sum; \n"
" } \n"
"} \n";

void multiply(const unsigned long size, const float *matrixA, const float
   *matrixB, float *result) {
  boost::compute::device gpu = boost::compute::system::default_device();
  boost::compute::context ctx(gpu);
  boost::compute::command_queue queue(ctx, gpu);

  boost::compute::program program =
    boost::compute::program::create_with_source(kernelSource, ctx);
  program.build();
  boost::compute::kernel kernel = program.create_kernel("matrix_multi");

  size_t workGroupSize = gpu.get_info<size_t>(CL_DEVICE_MAX_WORK_GROUP_SIZE);
  unsigned int maxThreadsPerBlock2D = sqrt(workGroupSize);
  size_t numBlocks = 1;
  size_t numThreads = (size_t)size;
  if (numThreads > maxThreadsPerBlock2D) {
    numBlocks = ceil((double)numThreads / maxThreadsPerBlock2D);
    numThreads = maxThreadsPerBlock2D;
  }
  size_t localSize[2] = { numThreads, numThreads };
  size_t globalSize[2] = { numThreads*numBlocks, numThreads*numBlocks };
```

```
35
36   boost::compute::vector<float> a_dev(size * size, ctx);
37   boost::compute::vector<float> b_dev(size * size, ctx);
38   boost::compute::vector<float> result_dev(size * size, ctx);
39   boost::compute::copy(matrixA, matrixA+(size*size), a_dev.begin(), queue);
40   boost::compute::copy(matrixB, matrixB+(size*size), b_dev.begin(), queue);
41
42   kernel.set_args(size, a_dev.get_buffer(), b_dev.get_buffer(),
       result_dev.get_buffer());
43   queue.enqueue_nd_range_kernel(kernel, 2, 0, globalSize, localSize);
44   boost::compute::copy(result_dev.begin(), result_dev.end(), result, queue);
45   queue.finish();
46 }
```

Listing 4.8: Matrix multiplication using *Boost.Compute*.

Although *Boost.Compute* provides an object oriented API, most lower-level tasks, such as compiling the GPU kernel (lines 21–23) and calculating the launch sizes (lines 25–34) are still in charge of the application programmer. Even using the higher-level STL-like *Boost.Compute* API, memory management tasks, including allocation and copying data, must still be explicitly programmed. Instead, by using GSPARLIB high-level pattern API, these tasks are completely abstracted from the application programmer.

Thrust API offers a structured programming approach built over the CUDA API. Similar to *Boost.Compute* higher-level API, Thrust abstracts many of the GPU programming burden of procedural APIs, but still requires the programmer to explicitly manage host and device memory interactions and compute indexes for parallel execution.

In Listing 4.9 the core Mandelbrot computation is defined as a functor (lines 6–35) annotated with `__host__` and `__device__` CUDA keywords (line 15). Since Thrust does not support two-dimensional kernels, we must use a `counting_iterator` (line 40) and calculate the row-major two-dimensional indexes (lines 17–18). This application does not involve any data vector as input, so the only memory copy is the result vector. Copying the data from the device vector (which resides in the GPU) to the host vector is explicitly done in line 43. GSPARLIB abstract this data copy and does not require the kernel to be defined in a special structure such as the functor in Listing 4.9.

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/copy.h>
4  #include <thrust/for_each.h>
5
6  struct mandel_functor {
7    double init_a;
8    double init_b;
9    double step;
10   unsigned long dim;
11   unsigned long niter;
12   unsigned char *M;
13   mandel_functor(double _init_a, double _init_b, double _step, unsigned long
       _dim, unsigned long _niter, unsigned char *_M):
14      init_a(_init_a), init_b(_init_b), step(_step), dim(_dim), niter(_niter),
       M(_M) {};
```

```
15    __host__ __device__
16    void operator()(const signed int xy) {
17      const long i = xy / dim; //truncates result
18      const long j = xy - (i*dim);
19      if (i < dim && j < dim) {
20        double im=init_b+(step*i);
21        double cr;
22        double a=cr=init_a+step*j;
23        double b=im;
24        unsigned long k = 0;
25        for (k = 0; k < niter; k++) {
26          double a2=a*a;
27          double b2=b*b;
28          if ((a2+b2)>4.0) break;
29          b=2*a*b+im;
30          a=a2-b2+cr;
31        }
32        M[xy]= (unsigned char) 255-((k*255/niter));
33      }
34    }
35  };
36
37  void mandelbrot(const double init_a, const double init_b, const double range,
        const unsigned long dim, const unsigned long niter, unsigned char *M) {
38    double step = range/((double) dim);
39    thrust::device_vector<unsigned char> M_dev(dim * dim);
40    thrust::counting_iterator<int> iter(0);
41    thrust::for_each_n(iter, dim*dim,
42      mandel_functor(init_a, init_b, step, dim, niter,
       thrust::raw_pointer_cast(M_dev.data())));
43    thrust::copy(M_dev.begin(), M_dev.end(), M);
44  }
```

Listing 4.9: Mandelbrot set calculation using Thrust.

One of the biggest differences between GSPARLIB and the aforementioned state-of-the-art tools is the code portability. By offering CUDA and OpenCL support, as well as abstractions over lower-level kernel functions, GSPARLIB is able to permit the same source code to use both backends. Even in a worst case scenario, on which a complex application require specific kernels to be written specifically for CUDA and OpenCL, GSPARLIB offers portability of host code and permits the programmer to focus on business rules instead of hardware architecture details or lower-level procedural APIs. This is not possible by using *Boost.Compute* or Thrust.

SkePU 3 offers support for multi-core CPU (using OpenMP) and many-core GPU (using CUDA and OpenCL) backends. Its programming API is based on algorithmic skeletons and uses modern C++ features such as variadic templates, template metaprogramming, and lambdas [EK19]. Data copies between the main and device memories are (mostly) automatic by using the `Vector` and `Matrix` smart containers. However, the drawback of not being thread-safe disqualifies it from our stream processing applications perspective. Listing 4.10

shows a simple matrix multiplication example using SkePU 3. It is based in an example from the official repository[4].

```
#include <skepu>

void multiply(const unsigned long size, float *matrixA, float *matrixB, float
    *result) {
  auto kernel = skepu::Map<0>([](skepu::Index2D idx, const skepu::Mat<float> a,
    const skepu::Mat<float> b) {
    float sum = 0;
    for (unsigned long k = 0; k < a.rows; k++) {
      sum += a.data[k * a.cols + idx.row] * b.data[idx.col * b.cols + k];
    }
    return sum;
  });
  skepu::BackendSpec skepuBackend;
  skepuBackend.setDevices(1);
  kernel.setBackend(skepuBackend);

  // Since Matrix constructor expects a vector, we need to copy this values
  std::vector<float> a_vec(matrixA, matrixA+(size*size));
  std::vector<float> b_vec(matrixB, matrixB+(size*size));
  std::vector<float> result_vec(result, result+(size*size));

  skepu::Matrix<float> a_dev(size, size, a_vec);
  skepu::Matrix<float> b_dev(size, size, b_vec);
  skepu::Matrix<float> result_dev(size, size, result_vec);

  kernel(result_dev, a_dev, b_dev);

  result_dev.updateHost();
  memcpy(result, result_dev.getAddress(), sizeof(float)*size*size);

  result_dev.releaseDeviceAllocations();
}
```

Listing 4.10: Matrix multiplication using SkePU 3.

The GPU kernel is defined by using a C++ lambda in lines 4–10. Its parameters are the thread index (`idx`) and the two matrices to multiply (`a` and `b`). Inside the GPU kernel, the smart container `Matrix` is called `Mat`. The data in the smart container can be accessed inside the GPU kernel by using the `data` property. We define a backend specification in line 11 and set it to use a single GPU in line 12. The GPU kernel object is set to use the backend specification in line 13.

The SkePU's `Matrix` smart container only accepts a `std::vector` as a host reference and the application in Listing 4.10 work with raw pointers. Thus, we copy the data into a `std::vector` container in lines 16–18 before passing them to the `Matrix` smart container in lines 20–22. Then, we invoke the GPU kernel in line 24 and manually update the host data container after finishing the computation in line 26. The `updateHost` call copies the data from the device memory to the host memory, then we copy it back to the memory pointer address in line 27. Finally, we release the GPU memory resources by calling `releaseDeviceAllocations`

[4]https://github.com/skepu/skepu/blob/master/examples/mmmult.cpp

in line 29. Note that the programming interface is, in our opinion, more user-friendly than GSPARLIB as can been seen also in Table 4.2. However, it offers less flexibility to implement optimizations for the algorithm or architecture. Our conceptual layered approach allows the user to create new parallel patterns and implement new code optimizations. Therefore, SkePU 3 has limitations when used as a runtime library for other tools or systems.

We briefly analyze the physical Source Lines of Code (SLOC) added to the sequential version of the applications discussed above as a rough measure of the programmer's productivity using the aforementioned GPU programming APIs and GSPARLIB. Table 4.2 present a summary of the physical SLOC of the applications presented in the previous Listings. The percentages represent the increase in SLOC required to exploit GPU parallelism compared to the sequential version. The metrics were generated using David A. Wheeler's 'SLOCCount' [Whe16]. The next section will discuss the performance of these applications.

Table 4.2: Physical SLOC comparison between GPU programming APIs.

Application	*Sequential*	*CUDA (Runtime API)*	*OpenCL*	*Boost. Compute*	*Thrust*	*SkePU 3*	GSPARLIB
Vector sum	74	169	216	107	97	90	97
	–	+128%	+192%	+45%	+31%	+22%	+31%
Matrix multiplication	78	134	159	118	104	91	98
	–	+72%	+104%	+51%	+33%	+17%	+26%
Mandelbrot set	81	108	150	118	105	87	101
	–	+33%	+85%	+46%	+30%	+7%	+25%

The OpenCL and CUDA version presents the highest number of SLOC overall. On average, they require respectively 127% and 78% more SLOC with respect to the sequential versions. SkePU 3 and GSPARLIB require, on average, 15% and 27% more SLOC with respect to the sequential versions, respectively. They represent the lowest increase in SLOC.

The CUDA and OpenCL versions are lower-level APIs and thus require more lines of code. Tasks such as memory allocation, querying device properties, and calculating the grid size (number of threads and blocks to launch) must be done explicitly by the programmer. *Boost.Compute* and Thrust abstracts the task of querying device properties and calculating the grid size, however, the programmer must still allocate device memory. *Boost.Compute* also requires the programmer to create the GPU context and communication queues, which explains the extra lines of code required to implement the applications using this API. SkePU 3 leverage modern C++ features such as variadic templates and lambdas, which allow the programmer to define the GPU kernel using less lines of code. The programmer is also able to invoke the GPU kernel using the same syntax of calling a normal C++ function. SkePU 3 offers a complete toolchain, which includes a source-to-source compiler to parse the user code and provide a higher-level API. Thrust, *Boost.Compute*, and GSPARLIB offer a similar abstraction level because they are libraries and can not afford user-defined GPU kernels as C++ code.

4.7 Performance Considerations

This section presents a performance comparison between GSParLib's Pattern API and some of the state-of-the-art tools discussed in Section 4.1. We compare our library with tools providing a similar level of abstraction, i.e. with a structured parallel programming API to exploit GPU parallelism. Therefore, we did not include lower-level solutions, such as OpenACC and StarPU. Nevertheless, we included CUDA and OpenCL to act as baseline benchmarks since they are the *de facto* standards for GPU programming.

Although GSParLib's objective is to provide high-level abstractions to GPU programming, we seek to offer similar performance to applications programmed in lower-level APIs (such as CUDA and OpenCL). This means that these abstractions should not hinder the application's performance. Given that the main purpose of our library is to be integrated with another tool for multi-core parallelism and used in stream processing applications, we measured separately the three phases of the GPU algorithms: (a) initialization, which includes compiling the GPU kernel, creating of GPU context, querying device properties, and creating execution flows (CUDA streams or OpenCL command queues); (b) computation, which comprises allocating GPU memory and copying data between the host and device memories, as well as launching the GPU kernel; and (c) finishing, on which the GPU memory allocations are released and any resources acquired during initialization are freed. On a stream processing application, the initialization step would usually be performed only once while the other steps are repeated for every stream element.

Since none of the GPU programming APIs discussed in the previous sections are focused on stream processing applications, we perform the performance comparison using only data-parallel applications. In addition, we did not use the batching capabilities of our library as the other APIs do not have any similar feature. We will address the performance of the combination of stream and data parallelism in Section 5.6.

All the experiments were carried out on a single machine with a CPU Intel® Core™ I9-7900X @ 3.3 GHz (10 cores and 20 threads), 48 GB of RAM memory (3×16 GB DDR4 @ 2400 MT/s) and a Titan Xp GPU with compute capability 6.1 and 12 GB GDDR5X @ 2400 MHz of memory. The NVIDIA driver installed was the 450.51.05. The system was running on Ubuntu OS release 20.04 LTS (kernel 5.4.0-40-generic). All programs have been compiled using g++ 9.3 and the `-O3` compiler flag, except the Thrust and SkePU 3 (CUDA) versions, which must be compiled with `nvcc`. We used `nvcc` version 11.0.194 for these. The implementation used the CUDA driver API with runtime compilation (NVRTC) from the CUDA Toolkit v11.0, NVIDIA's OpenCL 1.2, SkePU 3[5], and *Boost.Compute* 1.72.0. We focused in traditional HPC metrics such as execution time and speedup to observe the application scalability and performance. The stacked bars in Figures 4.9, 4.10, and 4.11 represent the

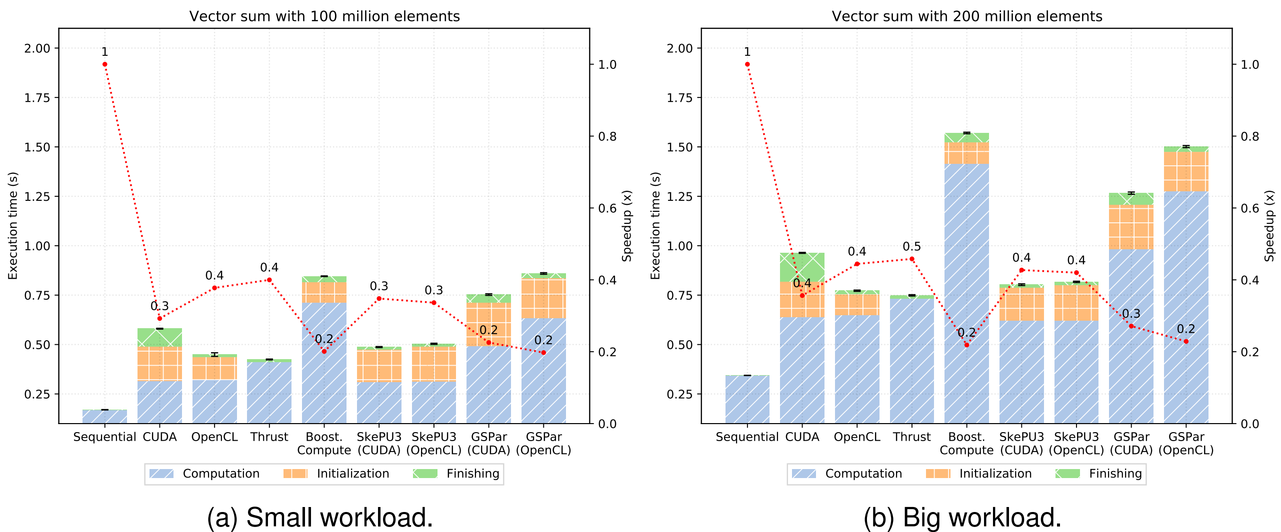
[5]https://github.com/skepu/skepu/commit/5a673cf89a131afc6c31cb6e65dbe7061c98d966

execution times in seconds of the three phases of the GPU applications and are related to the left Y axis. The Computation, Initialization, and Finishing phases are represented, respectively, by the blue, orange, and green bars. The total execution time is composed by the sum of the execution times of these three phases. The speedup with respect to the sequential version is presented as a red dotted line chart bound to the right Y axis. We ran each test 10 times and graph the average execution time. Standard error bars are shown in black color at the top of the columns and are related to the total execution time. The parallelism strategy used for implementing all the tested programs was the same across all APIs, thus the performance differences reported in the following tests are related to the abstractions and optimizations implemented on each API. It is important to highlight that API creators usually implement and test these optimizations based on the architectures they have access. Consequently, these optimizations may not make sense or are not so good when executing on different or newer GPU architectures. Differently, we have not implemented GPU kernel optimizations in GSParLib.

Our first tests are from a simple vector sum algorithm, which sums two vectors of 32 bits values. We tested two workloads: using 100 million elements (400 MB of data for each vector) and 200 million elements (800 MB of data for each vector). It employs the Map pattern to sum the two vectors and the Reduce pattern to sum the result vector into a single 32 bit value. The function signature used in all implementations is `unsigned long vecsum(const unsigned long vecsize, const unsigned long *a, const unsigned long *b, unsigned long *res)`. The expected output is that the `res` vector contains a summation of vectors `a` and `b` and that the function returns the summation of the values contained by the `res` vector. We overlapped the memory copy of the `res` vector from the GPU to host and the Reduce GPU kernel calls whenever possible. Figure 4.9 presents the performance results we obtained with this application. This pseudo-application presents a very low computing to global memory access (CGMA) ratio [KmWH16] and thus performs poorly in the GPU. In fact, all the GPU versions presented results worse than the sequential CPU version.

In CUDA (Listing 4.7) and OpenCL, we implemented the Map pattern as a single 1-dimensional GPU kernel call while the Reduce pattern is implemented using a tree-based approach [Har07], following the same structure described in Section 4.5.2. In Thrust, we defined the Map as a template *functor*, following the recommendations from the official documentation[6]. The *functor* is then called using Thrust's `for_each` and `make_zip_iterator` (combined with `make_tuple`) functions, while the Reduce is performed by calling `thrust::reduce`. We did not explicitly overlap memory copies and GPU kernel computation in Thrust due to the lack of methods to perform asynchronous copies in its API. In *Boost.Compute* we use the `transform` function to perform the Map and the `reduce` to perform the reduction. In SkePU we did not use the `MapReduce` skeleton for this task because it only returns the reduction result and we expect the summed vector to be returned as well. Therefore, we used

[6]https://docs.nvidia.com/cuda/thrust/#algorithms



(a) Small workload.

(b) Big workload.

Figure 4.9: Performance results of the vector sum algorithm using Map and Reduce.

two skeletons, `Map` and `Reduce`, to implement this pseudo-application. Finally, the GSPARLIB version was implemented using the `PatternComposition` class to combine `Map` and `Reduce` patterns, as described in Listing 4.6.

The speedups of Figures 4.9a and 4.9b are lower than 1, which highlights the memory bottleneck of this application. In fact, the NVIDIA Nsight Compute profiler reports an average of 85% of memory bandwidth occupation (the maximum ratio of achieved throughput with respect to the theoretical maximum throughput of any section of the GPU's memory system for compute, i.e. `gpu__compute_memory_sol_pct` metric) among all CUDA-based versions for the first GPU kernel (Map pattern), with Thrust achieving 90% of memory bandwidth occupation. In addition, the execution times of the GPU versions are mostly dominated by initialization tasks (except for Thrust, which does not have any initialization steps). The low time spent in the finishing step for the OpenCL versions may be related to the fact that the NVIDIA's OpenCL driver does not release resources when the release functions are called, as will be discussed in Section 4.8. Thrust present the best performance among the GPU versions in both workloads, however the OpenCL and SkePU versions presented very similar performance.

Our second pseudo-application is a naive matrix multiplication algorithm, which multiplies two square matrices of 32 bits elements manually linearized as a single vector. We tested three workloads, with matrices of 3,000x3,000 elements (each matrix accounts for 36 MB of data), 5,000x5,000 elements (100 MB for each matrix), and 10,000x10,000 elements (400 MB for each matrix). For each workload, we evaluate two ways to linearize the matrices: row-major layout and column-major layout [KmWH16]. The function signature used in all implementations is `void matmul(const unsigned long size, const float *a, const float *b, float *res)`. The execution time and speedups for these tests are presented in Figure 4.10. Since we aim at comparing the performance of the GPU programming

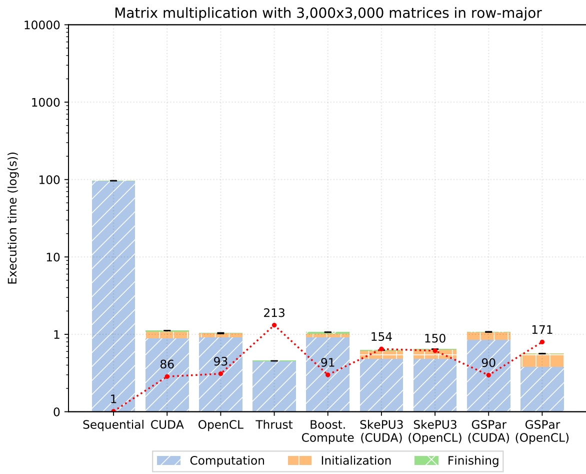
APIs, we did not employ any manual optimizations for the matrix multiplication algorithm. In fact, the core GPU kernel for all implementations is roughly the same for all versions, e.g.: `for(unsigned long k=0; k<size; k++) sum+=a[i*size+k] * b[k*size+j]; res[i*size+j] = sum;` for row-major layout. Therefore, our speedups are much lower that state-of-the-art results or specialized Single precision floating General Matrix Multiply (SGEMM) tools such as those implemented in Basic Linear Algebra Subprograms (BLAS) routines [FSH04, KDW10].

In CUDA and OpenCL, we implemented the matrix multiplication algorithm as a single 2-dimensional GPU kernel. In Thrust we defined a matrix multiplication *functor* which receives the matrices as parameters. As Thrust does not support bi-dimensional GPU kernels, we used a single `counting_iterator` from 0 to `5000*5000` (for the medium workload) and calculated the `i` (as `count/dim`) and `j` (as `count-i*dim`) variables inside the GPU kernel. In *Boost.Compute* we reused the same GPU kernel code from the OpenCL version and used the lower-level API to launch a 2-dimensional GPU kernel. The SkePU version was adapted from the examples provided by the authors[7]. One drawback of SkePU is that its standard `Matrix` container only accepts a `std::vector` as a host reference and our pseudo-application work with raw pointers. Therefore, we used a intermediate `std::vector` to pass the data between our raw pointer and the `Matrix` container. Nonetheless, this extra memory copies are performed in ~150ms for the big workload and represents less than 1% of the total execution time, thus it does not significantly impact the results. The GSPARLIB version was implemented as a single `Map` object with three input parameters (the matrices size and the `a` and `b` matrices) and a single output parameter (the result matrix). The GPU kernel core passed as argument in the `Map` constructor was the same for CUDA and OpenCL versions.

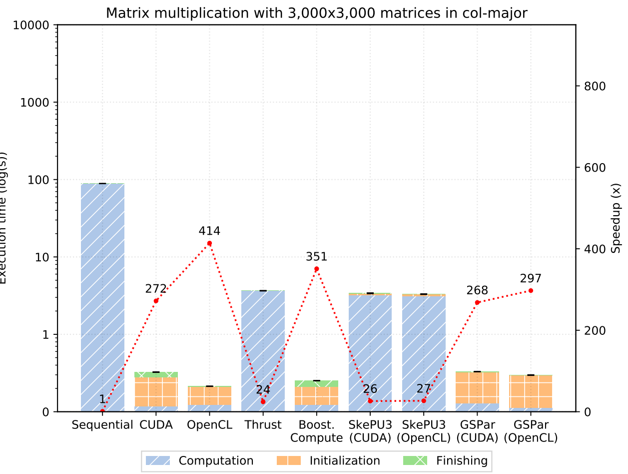
The most noticeable difference between the row- and column-major layouts are Thrust and SkePU performance results. They are clearly optimized to work with arrays ordered in row-major layout, which is the default for C, C++, and CUDA. This contrasts with CUDA, OpenCL, *Boost.Compute*, and GSPARLIB versions, which perform far better in column-major ordering because this simple matrix multiplication algorithm achieves coalesced memory access using this layout. In addition, for all workloads GSPARLIB present speedups on-par with the state-of-the-art tools. Moreover, GSPARLIB (OpenCL) version presented better performance results than pure OpenCL implementation for all workloads in row-major layout and for the big workload in column-major layout. More tests are needed to understand what optimizations explain these performance boost of GSPARLIB with respect to the lower-level library.

We used the NVIDIA Nsight Compute profiler to evaluate the CUDA-based versions of matrix multiplication with column-major layout in the medium workload (Figure 4.10d). We analyzed the device occupancy as the percentage of active warps to maximum warps per SM per active cycle (the `sm__warps_active.avg.pct_of_peak_sustained_active` metric) and the memory bandwidth occupation as the maximum ratio of achieved throughput with

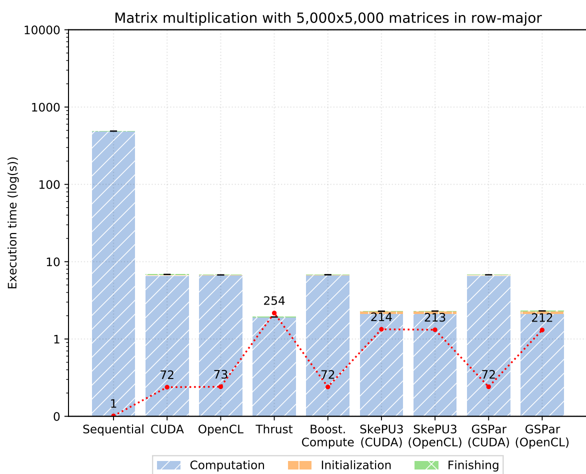
[7]https://github.com/skepu/skepu/blob/master/examples/mmmult.cpp



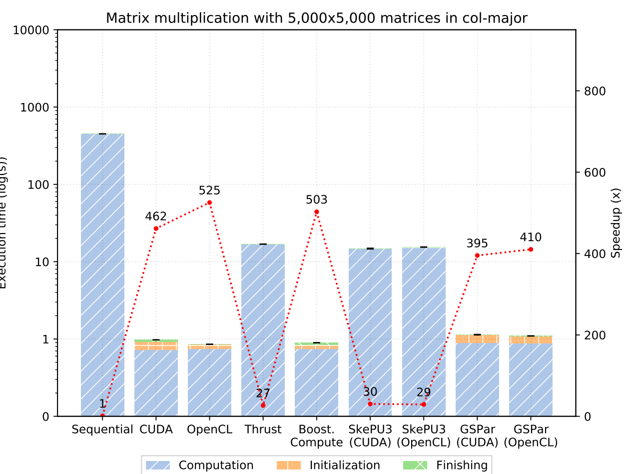
(a) Small workload in row-major.



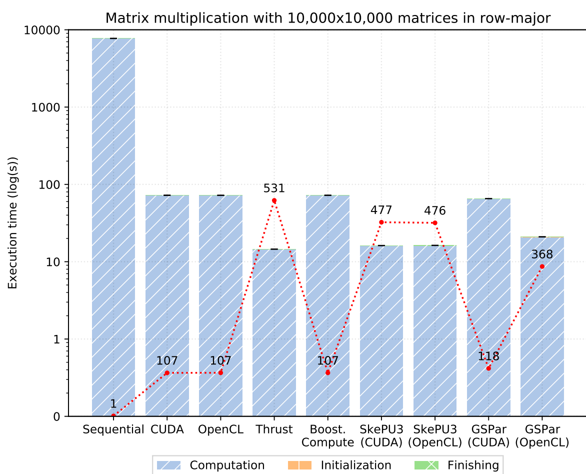
(b) Small workload in column-major.



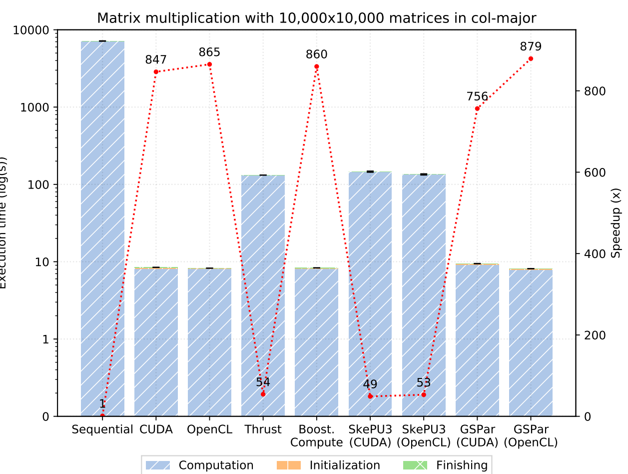
(c) Medium workload in row-major.



(d) Medium workload in column-major.



(e) Big workload in row-major.



(f) Big workload in column-major.

Figure 4.10: Performance results of the matrix multiplication algorithm using Map.

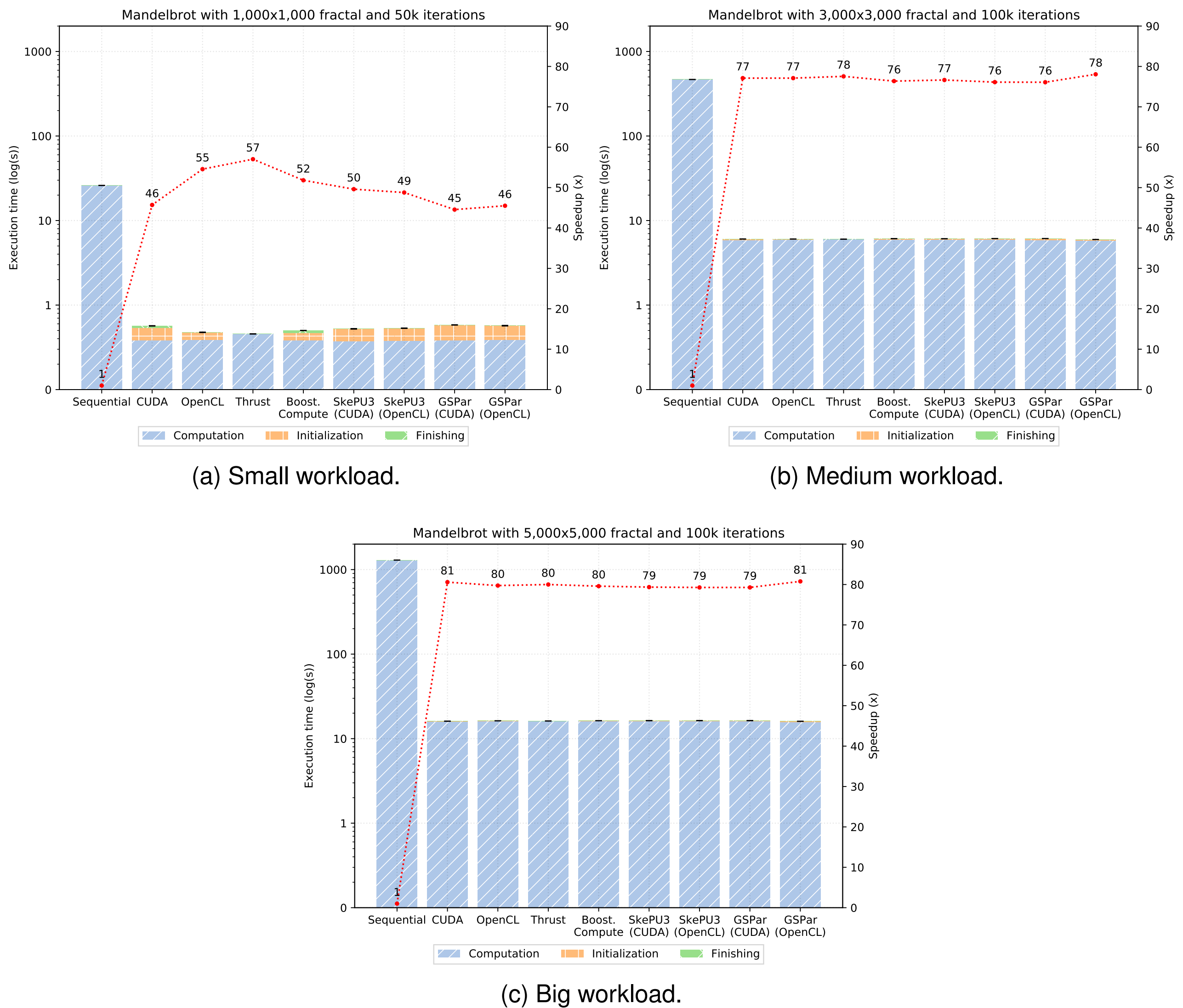
respect to the theoretical maximum throughput of any section of the GPU's memory system for compute (the `gpu__compute_memory_sol_pct` metric) [Bav19]. The profiler shows that

the CUDA version achieves 87% occupancy and 78% of memory bandwidth occupation, while GSParLib achieved 84% occupancy and 67% of memory bandwidth occupation. For comparison, Thrust achieves 74% and 89% of occupancy and memory bandwidth occupation, respectively, and SkePU 3 achieved 71% occupancy (the profiler was unable to calculate memory bandwidth occupation for the SkePU version). The profiler also demonstrated that Thrust and SkePU 3 launched the GPU kernels with 256 threads per block, while CUDA and GSParLib versions used 1,024 threads (the maximum threads per block of the Titan Xp GPU, according to `CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK`).

The third pseudo-application we tested was the Mandelbrot set calculation, which calculates the set of all points c in the complex plane that do not tend to infinity when the function $z \leftarrow z^2 + c$ it iterated up to a predefined limit [MRR12]. This workload is commonly used to evaluate how well each architecture performs floating point operations [BH10]. It is also useful to check load balancing algorithms because numbers outside the Mandelbrot set are calculated much faster since they quickly reach a predefined threshold that delimits numbers outside the set. In this cases, the algorithm does not need to perform all the iterations to decide that the number is outside the set and breaks the iteration loop sooner. However, for numbers inside the set the algorithm need to go through all the iterations to be sure that the number does not tend to infinity.

Our implementation of the Mandelbrot set (Listing 4.9 shows how we implemented this application using Thrust) generates a square fractal image where each pixel of the image represents a number between -2.125+-1.5 and 0.875+1.5. We tested three workloads: (a) 1,000x1,000 image with 50,000 iterations per number; (b) 3,000x3,000 image with 100,000 iterations per number; and (c) 5,000x5,000 image with 100,000 iterations per number. These workloads represent 1 million, 9 million, and 25 million numbers, respectively. The function signature is `void mandelbrot(const double a, const double b, const double range, const unsigned long dim, const unsigned long niter, unsigned char *M)`, where `a` is -2.125, `b` is -1.5, and `range` is 3. The `dim` parameter represents the image dimension (1,000, 3,000, or 5,0000) and `niter` is the number of iterations per number (50,000 or 100,000). The `M` pointer contains `dim*dim` bytes of memory in row-major order to hold the entire fractal image result and should be filled with a color representing whether the number is inside or outside the set. All computations use double precision floating point arithmetic.

Since the `M` vector is only for output, there is no memory copies into the GPU and the only global memory access of the GPU kernel is to write the result after performing all the calculations. Therefore, the data required to perform the computation is stored in registers. There is no overlap of memory copy and computation in this pseudo-application and the resulting data is synchronously copied to the host memory. All versions are implemented very similarly to the matrix multiplication implementation: CUDA and OpenCL uses a single 2-dimensional GPU kernel, Thrust uses a `counting_iterator` from 0 to `dim*dim` and calculates the `i` and `j` variables inside the GPU kernel, *Boost.Compute* uses the the lower-level API and



(a) Small workload.

(b) Medium workload.

(c) Big workload.

Figure 4.11: Performance results of the Mandelbrot set using Map.

the same GPU kernel code from the OpenCL version, SkePU uses as a single `Map` object with a lambda function, and the GSPARLIB version uses a single `Map` instance.

The performance results of this application are presented in Figure 4.11 and are similar among all the GPU programming APIs. In the small workload (Figure 4.11a), Thrust presented the best speedup of 57×, OpenCL presented 55× and CUDA presented 46× speedup. GSPARLIB versions presented 45× and 46× speedup for the CUDA and OpenCL versions, respectively. In the medium workload (Figure 4.11b) all versions presented speedups between 76× and 78×. The best speedup for this workload was presented by the Thrust and GSPARLIB (OpenCL) versions. In the big workload (Figure 4.11c) all versions presented speedups between 79× and 81×.

In summary, we conclude that GSPARLIB presented similar performance compared to state-of-the-art and lower-level APIs while offering a unified structured programming API and a driver-agnostic runtime. The results also revealed opportunities for implementing code

optimizations that leverage performance and maintain a similar abstraction level. Moreover, none of the state-of-the-art APIs offer features for stream processing applications, such as batching that makes GSPARLIB specially appealing for this kind of applications. This will be deeper demonstrated and investigated in Chapter 5, where it has been used as a runtime library for SPar's code generation. GSPARLIB can be seen also as a runtime library for exploiting GPU parallelism on APIs that aim higher-level abstractions.

4.8 Final Remarks

In this chapter we described GSPARLIB, a new library with a structured parallel programming API for GPU programming focused in stream processing applications. GSPARLIB offers a layered API that may suffice both experienced and novice programmers, with high-level yet powerful abstractions for exploiting GPU parallelism in multithreaded programs. We presented a comprehensive comparison of this new library with state-of-the art tools in terms of objectives, programmability, and performance aspects.

The single major contribution of GSPARLIB is to offer a unified structured programming interface and a driver-agnostic runtime that allow programmers to switch between the *de facto* standards CUDA and OpenCL backends simply by using a compiler flag. Moreover, it does not require the use of a custom compiler, providing the full functionality as a single dynamic library.

The importance of supporting both CUDA and OpenCL backends is highlighted by the low priority that NVIDIA seems to dedicate to its OpenCL driver, mainly considering that the company is the single biggest market player in the GPU segment. The third version of the OpenCL specification was released by Khronos in 2020 [Khr20b], while NVIDIA's OpenCL driver is still stuck in the eight years old OpenCL v1.2 [Khr20a]. Apart from the lack of updates, the current NVIDIA's OpenCL driver presents some unexpected behavior, such as not releasing the device memory when the `clReleaseMemObject` function is called if there is still any other unreleased OpenCL objects[8]. We faced this issue during the development of the library, which was specially challenging to overcome since we reuse the same `cl_program` for multiple GPU kernels and the driver does not release the device memory until this object is released with the `clReleaseProgram` function. Moreover, we also found some unexplained freezing when using the `clWaitForEvents` function from NVIDIA's OpenCL driver for multithreaded applications with many GPU kernels. The same application had no issues with the CUDA driver.

In this first version of GSPARLIB, we focused in the API design, trying to provide a good trade-off in programming freedom and abstraction level. We applied some optimizations (such as the parallel reduce discussed in Section 4.5.2), but further optimizations are left as

[8]https://bloerg.net/2013/01/15/opencl-resource-management.html

future work. Our performance tests are limited to a single NVIDIA GPU, therefore, more tests using multi-GPU and different accelerator makers such as AMD are also left as future work. Future works may also extend the GSPARLIB API to offer support for other parallel patterns.

5. HIGH-LEVEL STREAM AND DATA PARALLELISM FOR GPU WITH SPAR

In this chapter, we describe our extension of SPar language to express data parallelism along with stream parallelism. In Section 5.1 we introduce novel SPar attributes and their semantics. Their syntax is formally described in Section 5.2. Our extension did not change substantially the original syntax and semantics. In fact, the original stream capabilities of SPar were not modified since we only add novel attributes related to data parallelism. We also present a discussion on how to use these new attributes in Section 5.3.

The novel definitions and parallel pattern-based transformation rules targeting stream and data parallel patterns are presented in Section 5.4. We outline the differences with respect to the original definitions and rules defined by Griebler's work [Gri16]. The implementation of these new rules, focused in Map and Reduce parallel patterns are described in Section 5.5. Section 5.6 presents a discussion of the performance impacts of our implementation in three applications: Mandelbrot Streaming (5.6.1), lane detection (5.6.3), and ray tracing (5.6.2). In Section 5.7 we discuss the performance difference between our implementation in the SPar compiler for automatic stream and data parallelism compared to handwritten code of the same applications discussed in the previous section. Section 5.8 discusses the programmability of the SPar language using the new attributes. Finally, Section 5.9 present our final remarks about our extension of SPar language and compiler implementation.

5.1 Extending SPar Language

The current SPar attributes, presented in Section 2.5, are closely related to the stream parallelism domain. Also, they do not express any semantics of the data parallelism properties. In this work we focus in the widely used Map and Reduce patterns. However, in order to safely generate the Map pattern, the programmer must be sure that the operation being applied to the data elements can be executed in parallel, i.e. it is a *pure* function: "whose output depends only on its input and does not modify any other system state" [MRR12]. Functional programming semantics defines a pure function as "a function that, given the same input, will always return the same output and does not have any observable side effect" [LB20]. Since there is no standard way of automatically detecting this property in a given C++ code block [GP92, PCR12, BJB+20], the application programmer must provide this information.

None of the current SPar attributes provide information about the *pureness* of the code, thus we created a novel attribute called `Pure` to identify operations that can be safely executed in parallel [RGDF19]. The `Pure` attribute indicates that the annotated code block

is a pure function. This attribute may be used along with the `Stage` attribute list to mark the entire `Stage` as pure, or as an identifier attribute inside code regions annotated with `Stage` to mark specific portions of the `Stage` region as pure operations. The input and output data of the pure region are defined by the `Input` and `Output` attributes. In SPar, a `Stage` or code block is considered a pure function when it satisfies the following statements to guarantee correct use and correct code generation:

1. The `Pure` region can not have any side effects (i.e., mutation on non-local variables).
2. `Pure` loop iterations can not have execution order dependency (i.e., depending on the values modified by previous iterations).
3. The `Pure` region can not access any global variable that are not listed in the `Input` attribute.

From the programmer perspective, the `Pure` attribute is another attribute allowing to identify data parallelism inside the `Stage`. On the other hand, the compiler transformation rule identifies that this region/function can be computed in parallel over multiple data. It is up to the compiler decide which parallel architecture (GPU or multi-core) generate the stream parallelism with data parallelism code. Section 5.4 will describe the design of the compiler transformation rules to target data parallelism for GPUs.

Moreover, many potentially pure regions perform the accumulation of the computed values, which is an operation that combines the elements of a collection into a single element using a combiner operator. In the structured parallel programming jargon, this process is defined as the Reduce pattern [MRR12]. The combination of the input values is parallelizable as long as the operator is associative. In sequential applications, the accumulation is usually performed into a single output memory space, which is an atomic operation and cannot be directly parallelized. To allow the use of the `Pure` attribute in such code regions, the user must be able to indicate where this kind of operation is performed so that the compiler can introduce a parallel Reduce pattern when transforming the annotated source code.

For the current SPar attribute set, there was no way to express reduce operations. Therefore, we created the `Reduce` attribute to identify reduction operations. This new attribute increases the language expressiveness and allows `Pure` to be used in loops with atomic reduction operations, which would otherwise invalidate the pureness of the code. For the current version of SPar, we only allow `Reduce` to be used inside a `Pure` region, annotating a single statement which performs the reduction of a vector using a binary associative and commutative operator.

In our previous work, we evaluated different parallel programming models when implementing stream and data parallelism combined [RSG+19]. One lesson learned is that fine-grained stream processing may not generate enough workload to properly exploit massively parallel architectures such as GPUs. Thus, some stream processing applications

may not provide the expected performance scalability when using GPUs. For these cases, we are providing the possibility to express stream batches in SPar through the new auxiliary attribute for the `Stage`, named `Batch`, which activates the batching optimization for this specific `Stage` [HSS+14]. The programmer can specify as argument the size of the batch with literal or integer variable. In principle, this is the amount of stream items to be computed at once by the annotated stage, which must be a `Pure` stage. In short, `Batch` will now allow programmers to define the stream item granularity.

Observe that none of these attributes are related to the underlying parallel architecture. They were intentionally designed to express data parallelism properties such as data granularity (`Batch`), single instruction for multiple data (`Pure`), and data reduction (`Reduce`). If we compare to existing data parallel programming models such as OpenMP [DM98], `Batch` has a similar meaning to OpenMP *chunk*, `Pure` has a similar meaning to OpenMP *parallel for* where every computation inside the region can be performed in parallel and independently, and `Reduce` has the same meaning of OpenMP *reduction*.

For this work, data parallelism will be purposely exploited in GPUs. However, these new attributes are also open for further investigations and research on multi-core and cluster parallel architectures. The central point is that the programmer is no longer obliged to reason about the parallel architecture details when developing its application such as required by CUDA or OpenCL. SPar's compiler and transformation rules have to handle this complexities in place of programmers through its high-level annotation-based language.

Griebler and Fernandes [GF17] already proposed a modified syntax to express vector and array sizes in the SPar `Input` and `Output` attributes when studying distributed parallel support in SPar language. It was furthermore implemented in [Pie20]. This modified syntax allows the programmer to provide extra information to the compiler, which can apply optimizations based on the amount of data being transferred. Therefore, we implemented the modification proposed in [GF17], on which the user must inform the amount of elements of the array or vector in the `Input` or `Output` attributes with this syntax:

```
int size=10;
float statdata[10];
int *dyndata = new int[size];
[[spar::Pure, spar::Input(statdata[10]), spar::Output(dyndata[size])]]
```

Where `statdata` and `dyndata` are the variable names for two arrays with static and dynamic allocation, respectively. The static data has a fixed size of `10` elements while `size` represents the amount of memory allocated for the dynamic allocation. By using this syntax, the SPar compiler will be aware of how much memory is needed to be allocated and moved for each array or vector. This special syntax is not required when declaring variables of primitive types in `Input` and `Output`. Differently from [GF17], we do not enforce specific data types, however, the declaration of custom types must be accessible by the compiler.

It worth noting that this special syntax is not related to the hardware architecture and does not convey lower-level hardware details. We follow the standard C++ syntax and do not require the programmer to reason about lower-level concepts such as byte alignment and the amount of memory used by the variable. Instead of this, only the number of elements in the container. For now, the programmer is not required to inform the number of elements in the containers for `Input` or `Output` attributes, i.e. the new syntax is optional. We maintain backwards compatibility. However, future SPar versions may require this information in order to ensure the generation of optimized parallel code.

5.2 Syntax of the New SPar Attributes

In this section, we present the syntax of the novel SPar attributes introduced in the previous section: ***Pure***, ***Batch***, and ***Reduce***. We extend the syntax used by [Gri16] to introduce the original SPar attributes, which is based on the International Standard [Int17]. To distinguish the contribution of each work, the grammar defined by the C++ standard are presented in black colour while the grammar defined by [Gri16] are highlighted using the blue color, and the grammar introduced by this work are highlighted using the green color.

Griebler [Gri16] defines *aux_attr_list* as the list of auxiliary attributes that may be used together with the ***Stage***. Since our novel ***Pure*** and ***Batch*** attributes may also be used as auxiliary together with ***Stage***, we must redefine this term as follows:

aux_attr_list:

aux_attr_specifier aux_attr_list$_{opt}$

aux_attr_specifier:

, input_specifier

, output_specifier

, replicate_specifier

, pure_and_batch_specifier

NOTE: *Semantically, every auxiliary attribute may appear at most once in each list of auxiliary attributes following an identifier attribute.*

We do not enforce any special ordering to the auxiliary attributes. However, because the ***Batch*** must only be used together with ***Pure***, we define a *pure_and_batch_specifier* as follows:

pure_and_batch_specifier:

, pure_specifier_aux

, pure_specifier_aux aux_attr_list$_{opt}$ *, batch_specifier*

, *batch_specifier aux_attr_list*$_{opt}$, *pure_specifier_aux*

To use ***Pure*** as an auxiliary attribute together with ***Stage***, we define a *pure_specifier_aux*. When ***Pure*** is used as an auxiliary attribute, it marks the entire ***Stage*** code section as a pure function.

pure_specifier_aux:

pure_attr_aux

pure_attr_aux:

pure_token

pure_token:

pure_scoped_token

pure_scoped_token:

attribute_namespace :: ***Pure***

attribute_namespace:

spar

NOTE: *by definition,* ***Pure*** *attribute may have arguments, which is not currently supported.*

For many complex applications, the pure data-parallel region is nested within the ***Stage*** code region. Thus, we permit the use of the ***Pure*** as an ID attribute inside the ***Stage*** code region. The ***Pure*** as ID accepts the ***Input*** and ***Output*** attributes as auxiliary attributes. To this end, we define a *pure_specifier_id* as follows:

pure_specifier_id:

pure_annotated_iteration_statement

pure_annotated_iteration_statement:

pure_attr_id pure_iteration_statement

pure_iteration_statement:

for (*simple_declaration relational_expression* ***;*** *assignment-expression* ***)*** *statement*

pure_attr_id:

[[*pure_token pure_aux_attr_list*$_{opt}$ ***]]***

pure_aux_attr_list:

, *input_specifier*

, *output_specifier*

, *input_specifier* , *output_specifier*

, *output_specifier* , *input_specifier*

The ***Batch*** attribute must be used exclusively as auxiliarly together with ***Stage*** and ***Pure***. Similarly to the ***Replicate*** attribute, the ***Batch*** attribute also takes a single integer argument, which represents the batch size. However, unlike the ***Replicate*** argument, which

is optional, the ***Batch*** argument is required. A *batch_specifier* is used to define the ***Batch*** attribute:

batch_specifier:

  batch_attr attribute_argument_clause

batch_attr:

  batch_token

batch_token:

  batch_scoped_token

batch_scoped_token:

  attribute_namespace :: ***Batch***

attribute_namespace:

  spar

NOTE: *Semantically, exactly one argument is accepted to represent the batch size in a given stage. This argument can be an integer literal or an integer variable.*

Finally, the syntax of ***Reduce*** is defined to ensure a proper assignment expression and the use of an associative and commutative operator:

reduce_specifier:

  reduce_attr

reduce_attr:

  [[*reduce_token* ***]]*** *reduce_compound_statement*

reduce_compound_statement:

  { *reduce_assignment_expression* ***}***

reduce_assignment_expression:

  identifier reduce_assoc_comm_assignment_operator reduce_postfix_access_expr

  identifier = *identifier reduce_assoc_comm_operator reduce_postfix_access_expr*

reduce_assoc_comm_assignment_operator: one of

  *= += &= ^= |=

reduce_assoc_comm_operator: one of

  * + & ^ |

reduce_postfix_access_expr:

  identifier [identifier]

reduce_token:

  reduce_scoped_token

reduce_scoped_token:

  attribute_namespace :: ***Reduce***

attribute_namespace:

  spar

NOTE: *Semantically, the current version of SPar only accepts reduce_specifier to be used nested in a pure_iteration_statement.*

5.3 How to Annotate Sequential Codes with SPar

This section presents a programmer guide on how to use both the current and the new SPar attributes together for annotating stream and data parallelism in C++ sequential source code. The original SPar attributes were discussed in Section 2.5. The `ToStream` attribute is used to delimit the streaming region while the `Stage` attribute delimits a workstation in the assembly line. The `Input` and `Output` attributes define the data items sent between the stages. Finally, the `Replicate` attribute marks the `Stage` as stateless and indicates the degree of parallelism for this stage.

In the previous sections, we introduced three novel attributes for the SPar language: `Pure`, `Reduce`, and `Batch`. The `Pure` attribute is used as auxiliary with the `Stage` or as identifier inside the `Stage` region to identify pure functions. The `Reduce` attribute is used to mark atomic operations of data accumulation inside `Pure` regions. Finally, the `Batch` attribute is used as auxiliary together with `Stage` and `Pure` to enable the batching optimization for the stream processing application.When used together with the `Pure` attribute, the `Input` and `Output` attributes express the data dependencies of the pure function. For instance, if you have a `Pure` inside a `Stage`, you have to pay attention to which data is going to be consumed and produced inside this pure function. Since the `Reduce` attribute is used in a statement inside a pure function, you do not need to specify `Input` and `Output` dependencies for it.

Listing 5.1 exemplifies the use of the new `Pure` and `Batch` attributes together with the original SPar annotations. This example is computing the Mandelbrot Streaming application. `ToStream` (line 2) marks where the stream parallelism region starts and also refers to the stream generator stage. Inside the stream computation there are two `Stage` annotations identifying the stream operators. The data stream dependencies are specified through the `Input` and `Output` attributes. `Replicate` in line 5 indicates the degree of parallelism for that specific stage, running the amount of replicas given as argument in the attribute. The last `Stage` simply shows line by line the Mandelbrot image. It cannot be replicated because `ShowLine` is a stateful operator.

This Mandelbrot Streaming application was already presented in [Gri16], and Griebler took it from FastFlow examples repository in SourceForge. We are just adding the `Pure` attribute in the `Stage` annotation in line 5 of Listing 5.1 because the `for` loop in line 6 is a pure function. Moreover, we inserted the `Batch` attribute in line 5, allowing the control of the stream granularity for this specific `Stage`. It is worth point out that the application latency and throughput are directly impacted by the use of this attribute [HSS+14]. However, the programmer may test and choose the best configuration (size of the batch) that fits

the performance requirements. Section 5.6 will discuss the performance impacts of these attributes.

```
void mandel(int dim,int niter,double init_a,double init_b,double step) {
  [[spar::ToStream, spar::Input(dim, niter, init_a, init_b, step)]]
  for (int i=0; i<dim; i++) {
    unsigned char *img = new unsigned char[dim];
    [[spar::Stage, spar::Pure, spar::Batch(size), spar::Input(dim, niter,
   init_a, init_b, step, i, img[dim]), spar::Output(img[dim]),
   spar::Replicate(workers)]]
    for (int j=0; j<dim; j++) {
      double im = init_b + (step * i);
      double cr;
      double a = cr = init_a + step * j;
      double b = im;
      int k = 0;
      for (k=0; k<niter; k++) {
        double a2 = a * a;
        double b2 = b * b;
        if ((a2+b2) > 4.0) break;
        b = 2 * a * b + im;
        a = a2 - b2 + cr;
      }
      img[j] = (unsigned char) 255-((k*255/niter));
    }
    [[spar::Stage, spar::Input(img, dim, i)]] {
      ShowLine(img, dim, i);
      delete img;
    }
  }
}
```

Listing 5.1: Mandelbrot Streaming annotated with SPar using the new attributes.

To exemplify the use of the `Reduce` attribute, we present an application summing and multiplying the elements of multiple vectors in Listing 5.2. The vectors are streamed through the pipeline defined by the `ToStream` attribute in line 2. For each pair of vectors, the sum of the vectors is computed in line 12. After, the sum (line 13) and the product (line 14) of the elements in the resulting vector is computed using the addition and multiplication operators. This application just prints the result of the two reductions in the standard output. In this case, `Pure` is used as an identifier attribute, inside the `Stage` code region.

```
void vector_sum(int num_vectors, float vecsize, float **as, float **bs, float
    **results) {
  [[spar::ToStream, spar::Input(num_vectors, vecsize, as, bs, results),
   spar::Output(results)]]
  for (int v = 0; v < num_vectors; v++) {
    float *a = as[v];
    float *b = bs[v];
    float *result = results[v];
    [[spar::Stage, spar::Input(v, vecsize, a[vecsize], b[vecsize],
   result[vecsize])]] {
      float sum = 0;
      float product = 1;
      [[spar::Pure, spar::Input(a[vecsize], b[vecsize], sum, product),
   spar::Output(result[vecsize], sum, product)]]
      for (unsigned int i = 0; i < vecsize; i++) {
```

```
12          result[i] = a[i] + b[i];
13          [[spar::Reduce]] { sum += result[i]; }
14          [[spar::Reduce]] { product *= result[i]; }
15        }
16        std::cout << "Vector "<< v <<": sum is "<< sum <<" and product is "<<
       product << std::endl;
17      }
18    }
19  }
```

Listing 5.2: Vectors sums annotated with SPar using the `Pure` and `Reduce` attributes.

The special syntax for describing vector and array sizes is exemplified in line 5 of Listing 5.1, in which the image vector is specified as `img[dim]`. The vectors' sizes in Listing 5.2 are also specified in this special syntax in line 10. For the purposes of this work, these sizes define the amount of memory to be allocated and copied in the GPU memory.

5.4 New Compiler Transformation Rules for SPar

Currently, SPar supports the stream parallel patterns Pipeline and Farm. We aim to add support for the data parallel patterns Map and Reduce. Our focus in the Map pattern because it is the simplest widely used data-parallel pattern [MRR12]. Moreover, complex data-parallel patterns such as Gather, Stencil, and Scan, are just variants of the Map pattern. Operations that aggregate a set of data are also common in data-parallel applications, usually represented by critical sections inside the data-parallel region. These operations are represented by the Reduce pattern, which is commonly used together with Map.

Using functional semantics, we created our own definition for the data-parallel patterns Map and Reduce in order to identify which pattern will contain each code block from the annotated application. We defined the Map pattern as: $map(\square_{id}^{P})$, where $\square_{id}^{P}$ is the pure function or code wrapper that computes over multiple data independently. The input and output data can be a list, vector, or an array of data. Similarly, we defined the Reduce pattern as: $reduce(\square_{i}^{R})$, where $\square_{i}^{R}$ is a special code block containing a single statement which is a *reduce_compound_statement* defined in Section 5.2. The associative and commutative operator of the *reduce_compound_statement* is used to aggregate all the elements of a data set into a single element, which is produced by *reduce*. Furthermore, these two patterns can be combined to create a new pattern: $map\text{-}reduce(\square_{i}^{P}, \square_{i}^{R}, ..., \square_{n}^{R})$. In this case, one or more reduction operations are applied in the data elements produced by the Map pattern to aggregate them into a single output value.

We extend the original definition of the arguments of the Pipeline and Farm parallel patterns from [GDTF17] to support the combination of stream and data parallelism. The three *farm* components, namely emitter (E), worker (W), and collector (C), accepts as argument a single $\square_{id}$ or an instance of a data-parallel pattern (*map*, *reduce*, or *map-reduce*). We also

change the possible *pipe* arguments: each stage may be a $\Box_{id}$, a *farm*, a *map*, a *reduce*, or a *map-reduce*.

We present updated SPar definitions and novel transformation rules for the Map and Reduce parallel patterns. Before introducing our novel definitions and transformation rules, we extend the previous SPar notation (Section 2.5.1): P_i denotes a `Pure` attribute and Rdc_i denotes a `Reduce` attribute. We use $\forall_{id}\{\Box_{id}\}$ to denote a *pure_iteration_statement*, defined in Section 5.2, with the code block that is repeated for each iteration. The `Batch` attribute is not discussed in this section since it only changes the data management and does not interfere in the pattern generation. We present the details on how the `Batch` attribute interferes in the code generation in Section 5.5.

The original transformation rules for generating Pipeline and Farm parallel patterns from SPar annotations are based on six definitions, which are presented in Table 2.1. We extended these original definitions to support the transformation rules with the Map and Map-Reduce parallel patterns. Currently we only support the Reduce pattern when used together with Map, being therefore a Map-Reduce. The support for the standalone use of the Reduce pattern may be added in future works. Table 5.1 presents the new set of definitions with support to stream and data parallel patterns. The changes with respect to the definitions in [GDTF17] are highlighted using the green color.

From the original SPar transformation rules presented in Section 2.5.1, we take Rule 2.4 as an example to demonstrate the combination of stream and data parallelism. In this case, we are defining that the first stage is P where the code block is a $\forall\{\Box\}$ for Rule 2.4. Therefore, we apply D_1, D_{12}, and D_{13} to obtain Rule 5.1. In this case, we combine the Map and Pipeline patterns. Each stream item produced by the first *pipe* stage will be the input of the next stage that is instantiating the *map* pattern to exploit data parallelism.

$$[[T_0]]\{\Box_0, [[S_0, P_0]]\{\forall_0\{\Box_1^P\}\}\} \Rightarrow pipe(\Box_0,\ map(\Box_1^P)) \tag{5.1}$$

Similarly, if we take Rule 2.3 from Section 2.5.1, add P and consider a $\forall\{\Box\}$ as the code block of the first S. Consequently, we can apply D_1, D_5, and D_{11} to obtain Rule 5.2. In this case, a new parallel pattern is generated, combining Farm with the workers instantiating the Map pattern.

$$\begin{gathered}[[T_0]]\{\Box_0, [[S_0, O_i, R_n, P_0]]\{\forall_0\{\Box_1^P\}\}, [[S_1]]\{\Box_2\}\} \\ \Downarrow \\ farm(E(\Box_0),\ W(map(\Box_1^P)),\ C(\Box_2))\end{gathered} \tag{5.2}$$

Adding P in the Rule 2.5, with $\forall\{\Box\}$ as the code block, results in Rule 5.3. This Rule combines three parallel patterns: Pipeline, Farm, and Map, which are generated based on

Table 5.1: Definitions for transformation rules adapted from [GDTF17]. The definitions are applied in the order in which they are defined.

D_0	A *generic code block* ψ is generated for gathering results when the last $\square$ is annotated with S containing in its attribute list R_n and O_i.
D_1	A $\square$ becomes an argument of *map* pattern when it does not contain a Rdc_i and the first statement is a $\forall$ annotated with a S containing P in its attribute list.
D_2	A $\square$ becomes an argument of *map-reduce* pattern when the first statement is a $\forall$ annotated with a S containing P in its attribute list and the $\square$ contains a $\square^R$ which is annotated with Rdc_i.
D_3	A $\square$ can be the argument of a *pipe* pattern stage, or of a E or C in a *farm*, when its S annotation list does not contain the R_n attribute and D_1 and D_2 do not apply.
D_4	A $\square$ becomes an argument of W in a *farm* pattern when it is annotated with S containing an R_n attribute and D_1 and D_2 do not apply.
D_5	A *map* pattern becomes an argument of the W in a *farm* pattern when D_1 applies on a $\square$ and this $\square$ is annotated with S containing an R_n attribute.
D_6	A *map-reduce* pattern becomes an argument of the W in a *farm* pattern when D_2 applies on a $\square$ and this $\square$ is annotated with S containing an R_n attribute.
D_7	A $\forall\{\square^P\}$ which is annotated with only P inside a S annotation and does not contain a Rdc_i becomes a *map* pattern that will be nested into a stage of the *pipe* pattern or W in the *farm* pattern.
D_8	A $\forall\{\square^P\}$ which is annotated with only P inside a S annotation and contains a $\square^R$ annotated with Rdc_i becomes a *map-reduce* pattern that will be nested into a stage of the *pipe* pattern or W in the *farm* pattern.
D_9	A T becomes the *map* pattern when a $\square^P$ has $\forall_0$ as the first statement annotated with T and right after this $\forall_0$ there is only a single $\square^P$ which is a $\forall_1$ that does not contain a Rdc_i and is annotated with S containing P in its attribute list.
D_{10}	A T becomes the *map-reduce* pattern when a $\square^P$ has $\forall_0$ as the first statement annotated with T and right after this $\forall_0$ there is only a single $\square^P$ which is a $\forall_1$ that contains a $\square^R$ annotated with Rdc_i and is annotated with S containing P in its attribute list.
D_{11}	A T becomes a *farm* pattern when D_9 and D_{10} does not apply and the first S annotation contains R_n in the attribute list of two S at maximum.
D_{12}	A T becomes a *pipe* pattern when D_9 and D_{10} does not apply and the first S does not have R_n in the attribute list or when there are more than two S annotations.
D_{13}	A *map* pattern becomes an argument of a stage for the *pipe* pattern when D_1 applies on a $\square$, the S which annotates this $\square$ does not contain an R_n attribute, and D_{12} applies on the T that contains this S.
D_{14}	A *map-reduce* pattern becomes an argument of a stage for the *pipe* pattern when D_2 applies on a $\square$, the S which annotates this $\square$ does not contain an R_n attribute, and D_{12} applies on the T that contains this S.
D_{15}	A *farm* pattern becomes a stage for the *pipe* pattern when D_9, D_{10}, and D_{11} does not apply and $\square$ is annotated with S that contains R_n in the attribute list.

the following definitions: (a) D_1 to generate the *map* pattern and D_5 to it become an argument of the *farm*'s worker stage (W); (b) D_{12} to generate the *pipe* pattern from the T annotation; (c) D_{15} to generate the *farm* pattern as a *pipe* stage; and

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n, P_0]]\{\forall\{\square_2^P\}\}\} \\ \Downarrow \\ pipe(\square_0,\ farm(E(\square_1),\ W(map(\square_2^P))))\end{gathered} \tag{5.3}$$

We allow P to be employed as ID attribute, which provides more flexibility to SPar applications. If only part of the last `Stage` from Rule 5.3 is a pure function, P could be applied in this specific code block, as demonstrated by Rule 5.4. In this case, D_7 is applied to generate the *map* pattern nested in the *farm* pattern. In Rule 5.4, we defined a generic code block $\square_x$ to group a sequence of code blocks and patterns. This transformation phase allows us to apply D_4, where $\square_x$ is the argument of the W component in the *farm* pattern.

$$\begin{gathered}
[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, O_i, R_n]]\{\square_2, [[P_0]]\{\forall\{\square_3^P\}\}, \square_4\}, [[S_2]]\{\square_5\}\} \\
\Downarrow \\
\square_x = \{\ \square_2, map(\square_3^P), \square_4\ \} \\
pipe(\square_0,\ farm(E(\square_1),\ W(\square_x),\ C(\square_5)))
\end{gathered} \tag{5.4}$$

In Rule 5.5, we apply D_9 to generate a single *map* pattern from a T annotation schema due to the `Pure` attribute in this specific code structure.

$$[[T_0]]\{\forall_0\{[[S_0, P_0]]\{\forall_1\{\square_0^P\}\}\}\} \Rightarrow map(\square_0^P) \tag{5.5}$$

Rule 5.6 builds upon Rule 5.1 and applies D_2 to generate *map-reduce* based on the combination of the P and Rdc attributes. In this Rule, we defined the generic code block $\square_x$ to group the $\square_1^P$ and the $\square_2^R$ which are inside the $\forall_0$ code region. We can safely compute the reduction operation in parallel since: (a) the P attribute enforces that the region does not have any side effects nor depends on the execution order; and (b) the Rdc attribute enforces the use of a associative and commutative operator.

$$\begin{gathered}
\square_x = \{\ \square_1^P,\ [[Rdc_0]]\{\square_2^R\}\ \} \\
[[T_0]]\{\square_0, [[S_0, P_0]]\{\forall_0\{\square_x\}\}\} \\
\Downarrow \\
pipe(\square_0, map\text{-}reduce(\square_1^P, \square_2^R))
\end{gathered} \tag{5.6}$$

If we add the R_n attribute in the annotation schema from Rule 5.6, we obtain Rule 5.7. This Rule is also very similar to Rule 5.2, but now generating the *map-reduce* pattern instead of *map* (applying D_2 instead of D_1) and applying the D_0 to generate the generic collector. The generic code block $\square_x$ remains the same as in Rule 5.6.

$$\begin{gathered}\Box_x = \{\ \Box_1^P, [[Rdc_0]]\{\Box_2^R\}\ \} \\ [[T_0]]\{\Box_0, [[S_0, P_0, O_i, R_n]]\{\forall_0\{\Box_x\}\}\} \\ \Downarrow \\ farm(E(\Box_0), W(map\text{-}reduce(\Box_1^P, \Box_2^R)), C(\psi))\end{gathered} \tag{5.7}$$

The *map-reduce* pattern can also be generated when *Rdc* is inside a *P* used as ID attribute, as defined by D_8 and demonstrated by Rule 5.8. In this Rule, the generic code block $\Box_x$ remains the same as in Rules 5.6 and 5.7. In this case, the *map-reduce* pattern are in the same *pipe* stage as the code blocks before and after the *P* attribute ($\Box_1$ and $\Box_4$). Thus, we define a generic code block $\Box_y$ to group them in Rule 5.8. Multiple *map* and *map-reduce* patterns can be nested inside *pipe* and *farm* using this same logic.

$$\begin{gathered}\Box_x = \{\ \Box_2^P, [[Rdc_0]]\{\Box_3^R\}\ \} \\ [[T_0]]\{\Box_0, [[S_0]]\{\Box_1, [[P_0]]\{\forall\{\Box_x\}\}, \Box_4\}\} \\ \Downarrow \\ \Box_y = \{\ \Box_1, map\text{-}reduce(\Box_2^P, \Box_3^R), \Box_4\ \} \\ pipe(\Box_0, \Box_y)\end{gathered} \tag{5.8}$$

The *Rdc* attribute may also be nested inside the code block of the Rule 5.5, in which case the D_{10} is applied and Rule 5.9 is triggered. We demonstrate here the application of multiple *Rdc* attributes inside the scope of a single *P* annotation. Similarly to Rules 5.6, 5.7, and 5.8 we defined the generic code block $\Box_x$ to group the sequential code blocks which are inside the $\forall$ code region.

$$\begin{gathered}\Box_x = \{\ \Box_0^P, [[Rdc_0]]\{\Box_1^R\}, [[Rdc_1]]\{\Box_2^R\}\ \} \\ [[T_0]]\{\forall_0\{[[S_0, P_0]]\{\forall_1\{\Box_x\}\}\}\} \\ \Downarrow \\ map\text{-}reduce(\Box_0^P, \Box_1^R, \Box_2^R)\end{gathered} \tag{5.9}$$

These transformation rules represent in high-level how the annotated sequential code can be safely transformed into a parallel programming model specific code (PPMSC). The PPMSC is a source code with parallel constructs automatically generated by the compiler that is functionally equivalent to the sequential code [DDMT18]. The compiler generates the platform-specific PPMSC for the programming model chosen by the programmer by means of annotations or compiler flags.

As long as the programmer follows the syntax and semantic presented in Sections 5.1, 5.2, and 5.3, these rules also ensure the sequential equivalence of the generated code. Sequentially equivalent programs produce equivalent results (except for round-off errors) irrespective of the number of threads (one or many) used in its execution [MSM04].

5.5 Code Generation

We programmed the parallel pattern-based transformation rules (Section 5.4) in the SPar compiler generating the parallel patterns implemented by GSPARLIB Pattern API. These rules are applied before the original multi-core rules that uses FastFlow [GDTF17] or TBB [HGDF20] as runtime. However, to support GPU parallelism we enforce another requirement for the `Pure` attribute: **the `Pure` region (as well as the functions and structures referenced by it) should be valid CUDA C or OpenCL C99, according to the underlying driver**. We also limit the structure and types of loop statements that can be annotated with the `Pure` attribute according to the syntax presented in Section 5.2. Any of these limitations can be lifted by future works.

We added two compiler flags to the SPar compiler to control the behavior of the new GPU backend:

- `-spar_gpu`: this flag enables the new rules for generating the data-parallel patterns of GSPARLIB for the GPU backend;
- `-spar_opencl`: this flag enables the GSPARLIB OpenCL driver in the SPar compiler (it uses the GSPARLIB CUDA driver by default). Since the GSPARLIB Pattern API is driver-agnostic (as discussed in Chapter 4), when this flag is set the SPar compiler just changes the included file (`GSPar_OpenCL.hpp` instead of `GSPar_CUDA.hpp`) and the namespace used (`GSPar::Driver::OpenCL` instead of `GSPar::Driver::CUDA`) for referencing the `Instance` class in the generated code. This is a way for the programmer to choose when using OpenCL or CUDA on NVIDIA boards.

The first step performed by the SPar compiler after scanning and parsing the code (for a high-level representation of the full compiler flow, please refer to Figure 2.6) is matching the Transformation Rules in the AST to check which rules should be applied. Figure 5.1 illustrates an overview of the order on which the novel Transformation Rules presented in Section 5.4 are applied by the SPar compiler. For each `ToStream` annotation, we first check if the Transformation Rules 5.5 or 5.9 (which are based in the definitions D_9 and D_{10}) applies. This step is represented by the first decision ("Is pure data parallelism?") in the flow of Figure 5.1. In this case, if there is any `Reduce` attribute inside the code block, the Rule 5.9 is applied to generate the *map-reduce* pattern. Otherwise, the Rule 5.5 is applied to generate

the *map* pattern. When any of these two Rules are applied, we do not check for other rules on this `ToStream` since the entire region is replaced by a single Map (or Map-Reduce) pattern.

Then, we iterate over the `Stage` attributes inside the `ToStream` region. For each `Stage`, we first check if they have the `Pure` attribute as AUX. This step is represented by the second decision ("Is Stage Pure?") in the flow of Figure 5.1. In this case, if there is any `Reduce` attribute inside the pure code block we apply Rules 5.6 or 5.7. Otherwise we apply one of the Rules 5.1, 5.2, or 5.3, according to the annotation schema.

Finally, if the `Stage` auxiliary attribute list does not contain the `Pure` attribute, we check if the `Pure` attribute was used as ID inside the `Stage` region. This step is represented by the last decision ("Is there any Pure region inside Stage?") in the flow of Figure 5.1. In this case, we apply the Transformation Rules 5.4 (to generate the *map* pattern) or 5.8 (to generate the *map-reduce* pattern).

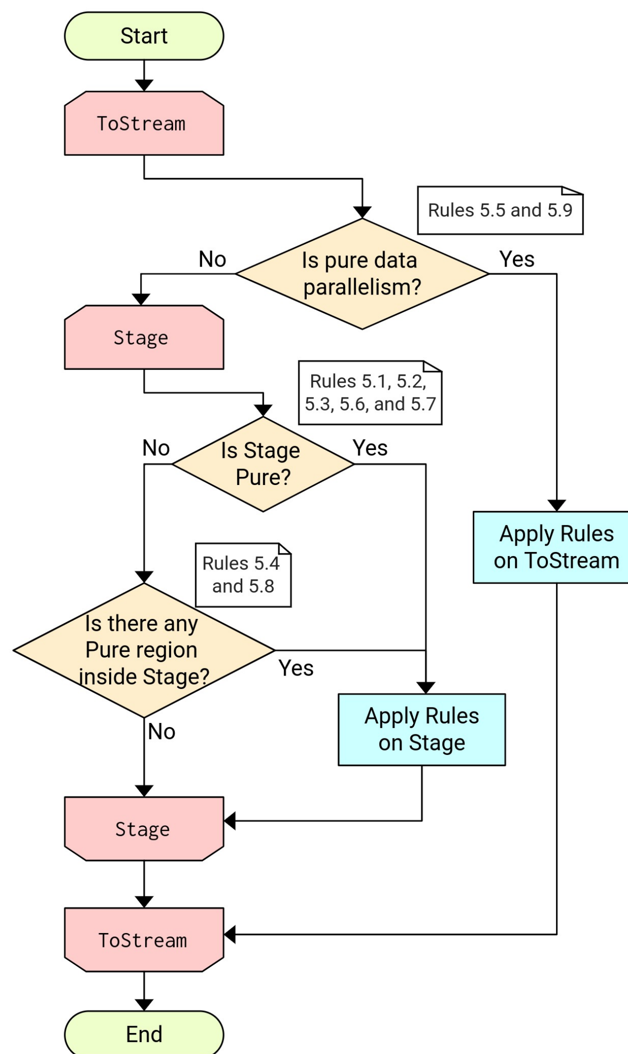


Figure 5.1: Flow of the new transformation rules presented in Section 5.4.

Once the Transformation Rules are identified and the compiler knows which parallel patterns should be generated, it starts the actual code generation targeting GSParLib data-parallel patterns. Firstly, if there is a `Reduce` attribute in the `Pure` block, the components of the

statement annotated with `Reduce` are parsed into the GSPARLIB's `Reduce` class constructor parameters: the input vector, the binary associative and commutative operator, and the output result. Then, this statement is removed from the body of the `Pure` region since the Reduce is computed after the Map.

The `for` statement annotated with `Pure` is parsed to identify the iterator variable name as well as the starting and ending expressions of the `Map`, which are used as the `min` and `max` values of the GSPARLIB's `Dimensions` structure. The iterator variable name is set into the `Map` class using the `setStdVarNames` method. If there is another `for` statement directly nested with the first `for`, i.e. if there is not any other statements between the two `for` statements, then we try to parse it as the second dimension for the `Map` class. After, the code block inside the `for` body is turned into a string and set as the `Map` class constructor parameter (the kernel core). We navigate this code block that represents the core of the GPU kernel to check if there are any `struct` or function called from the kernel. These definitions are then also marked for inclusion within the kernel code (using the `addExtraKernelCode` method from GSPARLIB API). The parameters for the `Map` class are parsed from the variables inside `Input` and `Output` annotations used together with `Pure`.

Additionally, if the `Pure` is used as AUX attribute together with `Stage`, we check if the `Batch` attribute is also present to set the batch size for the patterns accordingly. We only support `Batch` to be used together with `Stage` and `Pure` as AUX because the stage must prepare the batches of streamed data for the Map input, which is not currently possible if the `Pure` is used as ID attribute.

If the Rules 5.5 or 5.9 applies, we replace the entire `ToStream` region with a code block that perform all these steps, in the following order: creates the patterns instances, sets any extra definitions needed, sets the standard variable names, sets the parameters, run the patterns, and deletes the pattern instances. Since the entire `ToStream` is replaced by the Map or Map-Reduce pattern, the multi-core transformation is essentially disabled for this specific `ToStream`. In this case, the `Batch` attribute is ignored since there is no streaming to batch values for the data parallelism.

For the Transformation Rules that combine stream and data parallelism, we first include the GSPARLIB's data parallel patterns (Map and Map-Reduce) in the code and then let the multi-core transformation rules apply the Pipeline and Farm patterns. To this end, we insert a sequence of statements before the `ToStream` annotation in the original source code which: (a) prepare the `Dimensions` variable for the pattern; (b) create an instance of the identified patterns (using the `new` C++ keyword); (c) set the extra definitions for `structs` and functions used inside the kernel code; (d) set the standard variable names (iterator variable); (e) set the batch size (if the pattern is batched); (f) set the parameters as placeholders; (g) compile the pattern using the dimension declared in the first step. If there is a Reduce component, we also construct the `PatternComposition` (discussed in Section 4.5.3) instance to create the Map-Reduce pattern. This initialization steps are performed before the streaming

region so that they are performed only once for the entire execution. Since the pattern is already compiled with the dimensions, it stores this information and we do not need to pass it again when running the pattern, as described in Section 4.5.

After generating this code block, we add the `Map` or `PatternComposition` instance in the `ToStream` `Input` attribute arguments. Then, we insert a statement cloning the pattern right after the `ToStream` annotation, which represents the first Pipeline stage or the Farm's emitter. This cloned pattern is added as argument of the `Input` attribute of the `Stage` annotated with `Pure`. Since these rules are applied before the multi-core rules that generates the Pipeline and Farm patterns, adding the Map and Reduce objects in the `Input` automatically includes them in the structure that defines the stream items. Making a clone of the pre-compiled pattern for each stream item in the first stage also permits their use in a multi-threaded environment since the pattern objects are lightweight but are not thread-safe (as discussed in Section 4.5).

The next step during the code generation phase is to replace the original `Pure` region with a code block with sentences for: (a) setting the actual parameters; (b) calling the `run` method; (c) deleting the patterns objects. Finally, we generate statements to delete the original patterns after the `ToStream` scope. These steps are essentially the same whether the `Pure` attribute is used as AUX together with `Stage` or as ID inside the `Stage` scope.

After this transformations aiming data parallelism patterns using the GSPARLIB runtime, the multi-core transformation rules implemented by [Gri16] are applied. The only substantial change we made in the multi-core transformation rules are to support batched parallel patterns because the batching must be done in the streaming region. If the `Batch` attribute is present in a `Stage`, we add a vector of stream items in the stage structure to store all the incoming stream items. When the vector reaches the size defined as argument of the `Batch` attribute or if the stream comes to an end, the entire batch of items is processed at once using the batch support of GSPARLIB, presented in Section 4.5.1, and the items are sent to the next stage. This process increases the latency but improves throughput in cases where each stream item does expose enough parallelism to worth offloading the computation to the GPU. The application developer should consider this trade-off between throughput and latency to decide whether to use the `Batch` attribute and the batch size that best suits their needs. We will present details of this trade-off in Section 5.6.

5.6 SPar Performance Considerations

In this section we perform tests to evaluate the performance of the code generated by the SPar compiler targeting stream and data parallel patterns combined. There is a lack of standard benchmarks for combined stream and data parallelism in the academia. Thus, we used stream processing applications that also expose data parallelism, which allows us to combine the original SPar annotations (focused in stream parallelism) with our novel attributes

that are focused in data parallelism. This section shares many similarities with Section 4.7, such as: (a) we use the same environment of tests and the same test methodology; (b) in addition to the total execution time, we discuss streaming related metrics such as latency and throughput; and (c) initialization and finishing tasks are included in the total execution time, unlike the tests in Section 4.7 on which were measured separately. We used the original FastFlow runtime and the novel GSPARLIB runtime in the following tests. We calculate the standard error of mean (SEM) for each test using the formula $\sigma/\sqrt{n}$, where σ is the sample standard deviation and n is the number of measurements [BB12].

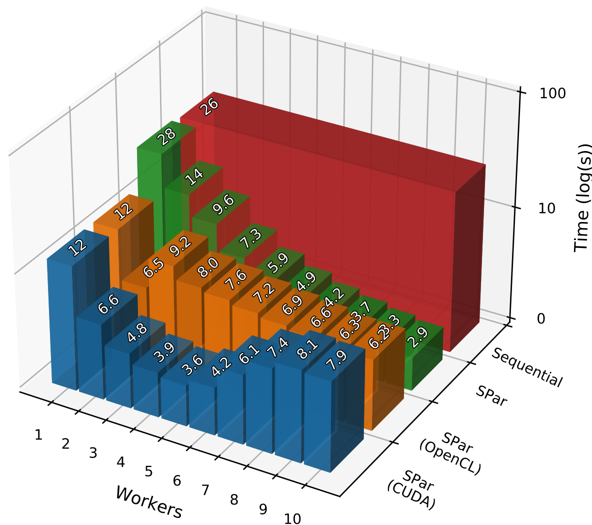
5.6.1 Mandelbrot Streaming

The standard Mandelbrot set calculation algorithm was already discussed as a data parallel application in Section 4.7. Here we test the modified version of the Mandelbrot algorithm that exposes stream parallelism presented in Listing 5.1. We test the same workloads used in Section 4.7, however, in this test each row of the image is a single stream item, which means that we are sending 1, 3, and 5 thousand stream items, respectively, for the three workloads. This application uses a library to show the fractal image in the screen (which was disabled for the performance tests). We checked that the images shown by the sequential and parallel versions were the same.

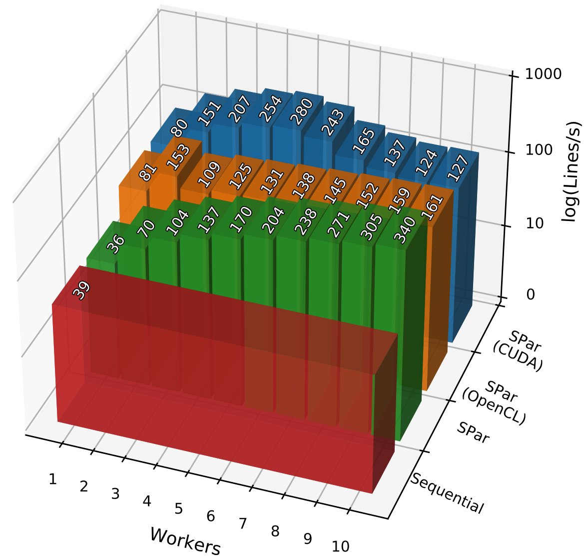
The charts in Figures 5.2, 5.3, and 5.4 present the total execution time and throughput, using base 10 logarithmic scale, for the three workloads. The Tables 5.2, 5.3, and 5.4 present the standard errors for each of these Figures. We calculate the throughput of the Mandelbrot Streaming application as the number of lines processed per second (i.e. *throughput = image dimension/execution time in seconds*) since they represent the data items that move through the stream. The number of parallel workers (1 to 10) is related to the X axis of the graphs while the different versions (sequential, multi-core SPar, and SPar using CUDA and OpenCL drivers of GSPARLIB as backend) are presented alongside the Y axis. It worth noting that the number of parallel workers in the X axis does not necessarily represent the number of active threads in the system. Rather than this, it represents the number of parallel workers in the replicated stage of the Farm pattern. There are other threads dedicated for the sequential stream stages (such as the emitter and the collector of the Farm pattern and any stateful stage of the Pipeline pattern). The values (not log-scaled) are shown at the top of each bar. For this first set of tests, we did not include the `Batch` attribute.

Figure 5.2 presents the execution time and throughput for the first (small) workload. The SPar version (green) present consistent performance improvements as the number of parallel workers increase. The performance of the SPar version with GSPARLIB using one and two parallel workers is very similar when using CUDA and OpenCL backends. However, the OpenCL version's performance degrades sharply when using three workloads

and improves gradually as the number of workers increase, reaching its peak performance using ten workers with a total execution time of 6.23 seconds, 4.2× speedup with respect to the sequential version, and throughput of 161 lines per second. The CUDA version's performance improves up to five workers, with a total execution time of 3.58 seconds, 7.2× speedup with respect to the sequential version, and throughput of 280 lines per second. Further increasing the number of parallel workers degrades the CUDA version's performance. We present the SEM in seconds in Table 5.2, which are all under 0.4 seconds.



(a) Execution time of small workload.



(b) Throughput of small workload.

Figure 5.2: Performance results of Mandelbrot Streaming for the small workload.

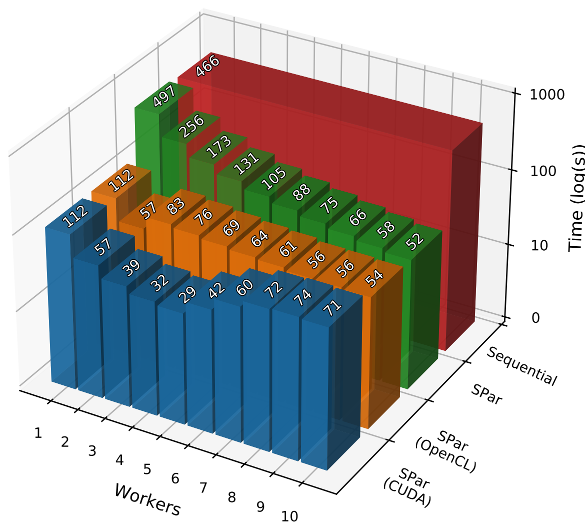
Table 5.2: SEM in seconds for tests of Figure 5.2a.

Name	*Workers* 1	2	3	4	5	6	7	8	9	10
SPar (CUDA)	0.047	0.020	0.122	0.033	0.071	0.157	0.133	0.314	0.125	0.140
SPar (OpenCL)	0.054	0.028	0.075	0.058	0.087	0.114	0.140	0.089	0.064	0.080
SPar	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Sequential	0.001									

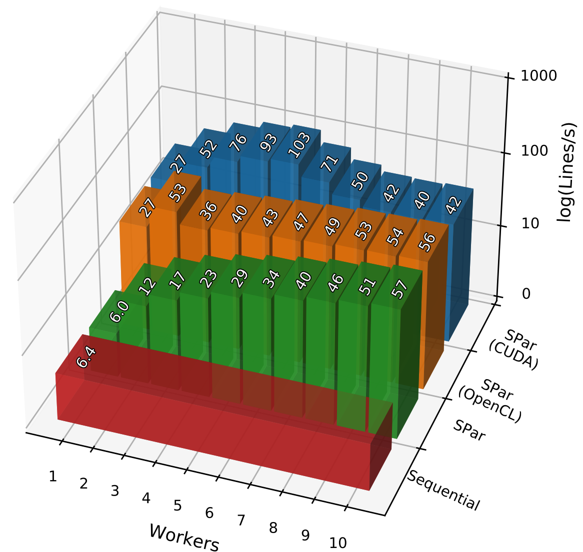
Further increasing the number of workers did not reflect on performance improvement as GPU is becoming the bottleneck of the application. There are multiple threads to offload the computations for a single GPU. Thus, with more workers, the high number of kernel invocations become the bottleneck of the application. For this workload, the multi-core surpasses the best GPU performance with nine and ten workers.

Figure 5.3 presents the execution time and throughput for the second (medium) workload. The results are very similar to those presented by the small workload and the peak performance is also achieved by SPar (CUDA) version at five workers (total execution time

of 29 seconds, 15.9× speedup regarding to the sequential version, and throughput of 103 lines per second). Although CUDA and OpenCL versions presented the same performance when using one and two worker threads, in its throughput performance peak, the CUDA version presents almost twice the throughput of the best OpenCL version: 103 lines/s (CUDA with 5 workers) compared to 56 lines/s (OpenCL with 10 workers). Actually, the OpenCL version present slight increasing performance as the number of parallel workers increase beyond two. Nonetheless, even using 10 parallel workers, the multi-core version presents worse execution time and throughput than the CUDA version for this workload. Since this workload presents a heavier load for each stream item, the massive parallelism of the GPU cover the costs of copying data to the GPU memory and invoking the kernel. The sequential and multi-core versions as well as the GPU versions with a single worker thread, presented standard errors under 0.1 s (shown in Table 5.3). The GPU versions with a higher number of threads presented higher SEM, topping at 1.12 s for the CUDA version with 7 workers.



(a) Execution time of medium workload.



(b) Throughput of medium workload.

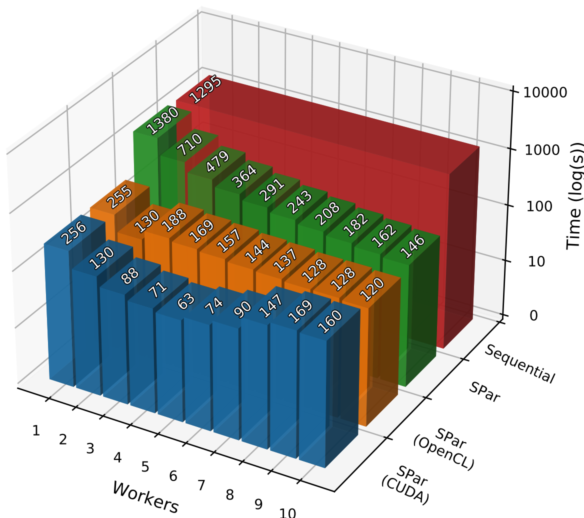
Figure 5.3: Performance results of Mandelbrot Streaming for the medium workload.

Table 5.3: SEM in seconds of tests for Figure 5.3a.

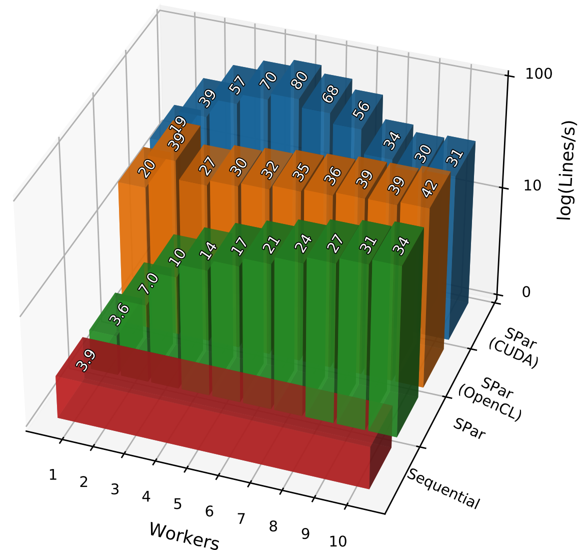
Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.006	0.087	0.089	0.162	0.377	1.077	1.119	1.074	0.754	0.532
SPar (OpenCL)	0.012	0.073	0.302	0.264	0.439	0.535	0.539	0.288	0.548	0.261
SPar	0.009	0.003	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Sequential	0.012									

Figure 5.4 presents the execution time and throughput for the third (large) workload. The overall behavior is still the same: the CUDA version achieves the peak performance

with five workers (total execution time of 62.5 seconds, 20.7× speedup with respect to the sequential version, and throughput of 80 lines per second). The OpenCL and CUDA performance are very similar using one and two workers, however, the OpenCL version struggles to utilize more parallel workers to boost the performance. Since each row now represents a bigger load (5,000 pixels), each stream item launches a bigger GPU kernel, which worth the extra overload involved in offloading the computations to the GPU. The standard errors of this test, presented in Table 5.4, also shown a similar behavior of the medium workload. The CPU versions and GPU versions using a single thread present standard errors well under 0.1 s, while the GPU versions using more threads present an increased standard error. The biggest standard errors, of 2.33 s and 2.28 s, are presented by CUDA with 7 and 8 workers, respectively.



(a) Execution time of large workload.



(b) Throughput of large workload.

Figure 5.4: Performance results of Mandelbrot Streaming for the large workload.

Table 5.4: SEM in seconds of tests for Figure 5.4a.

Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.003	0.148	0.092	0.228	0.708	1.354	2.329	2.283	1.286	1.052
SPar (OpenCL)	0.002	0.139	0.577	0.716	0.635	0.580	1.325	0.791	1.030	0.624
SPar	0.017	0.012	0.002	0.002	0.002	0.003	0.002	0.001	0.002	0.002
Sequential	0.022									

Nonetheless, for most stream processing applications, instant measurements such as instant latency are more meaningful than overall throughput. Thus, we measure the latency adding a timestamp to each stream item in the first stage and checking the time it spent in the last stage. Since the first stage is much faster than the other stages, we used the

on-demand scheduler of SPar to avoid measuring the time data that the item spend waiting in the worker's queue. We chose to discuss only the SPar multi-core version and the SPar CUDA version with the medium workload for the sake of space. Our tests suggest that the same conclusions hold true to the OpenCL version and to the other workloads as well.

Figure 5.5 presents a profiling to evaluate the impact of different numbers of parallel workers in the replicated stage in the latency of the medium workload for the SPar using the multi-core runtime (without GPU offloading). We present the profile of 1, 2, and 10 parallel workers for this application. There is no batching of data items in Figure 5.5, since the new `Batch` attribute is only supported with the GSPARLIB runtime and it must be used together with `Pure` attribute. Figure 5.6 presents a profiling for different numbers of workers and batch sizes in the latency of the medium workload for the SPar (CUDA) version. We present the profile using 1, 2, and 10 parallel workers in the replicated stage (each row of graphs in Figure 5.6 represent a different number of workers) as well as no batching and batch sizes of 2, 10, and 30 items (represented by each column of graphs).

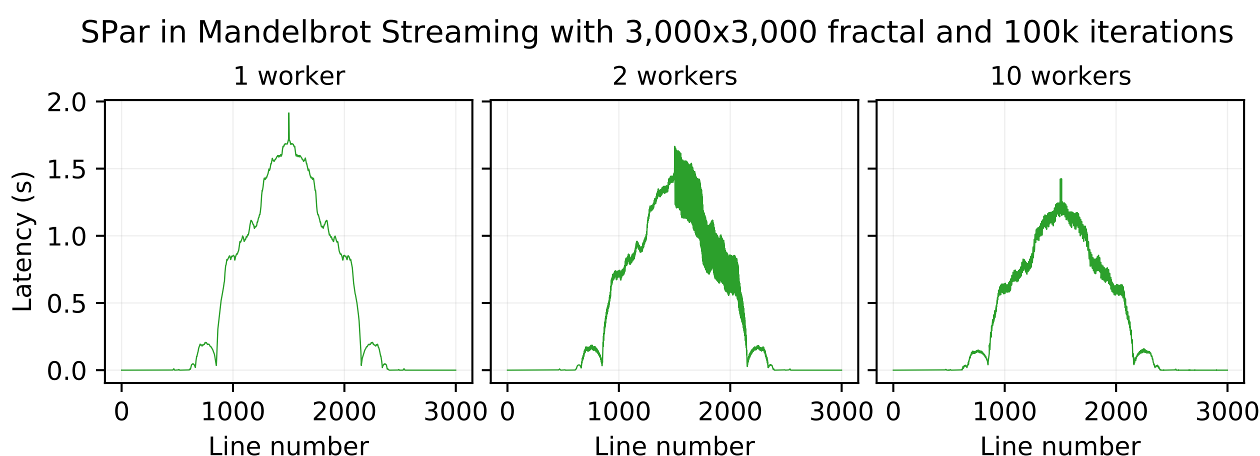


Figure 5.5: Latency profiling of SPar Mandelbrot Streaming with the medium workload.

Each stream item is a single line of the Mandelbrot image, therefore, we have 3,000 items for the medium workload. The first and last lines of each test present near-zero latency numbers, which is a characteristic of this application: numbers outside the Mandelbrot set are calculated much faster since they quickly reach the threshold that delimits numbers outside the set (line 15 in Listing 5.1). In this cases, the algorithm does not need to perform all the 100,000 iterations to decide that the number is outside the set and breaks the loop sooner. This explains the overall behavior of low latencies of the first and last lines, and the high latencies in the middle lines, where the algorithm goes through all the iterations to be sure that the number is on the Mandelbrot set.

We can see the influence of the number of workers on the latency of each item in the Figure 5.5 and in the first column of graphs of Figure 5.6 (no batching). The peak latencies of the SPar multi-core version without GPU offloading (Figure 5.5) are 1.92 seconds for 1 worker, 1.66 seconds for 2 workers, and 1.43 seconds for 10 workers. Nonetheless, the first

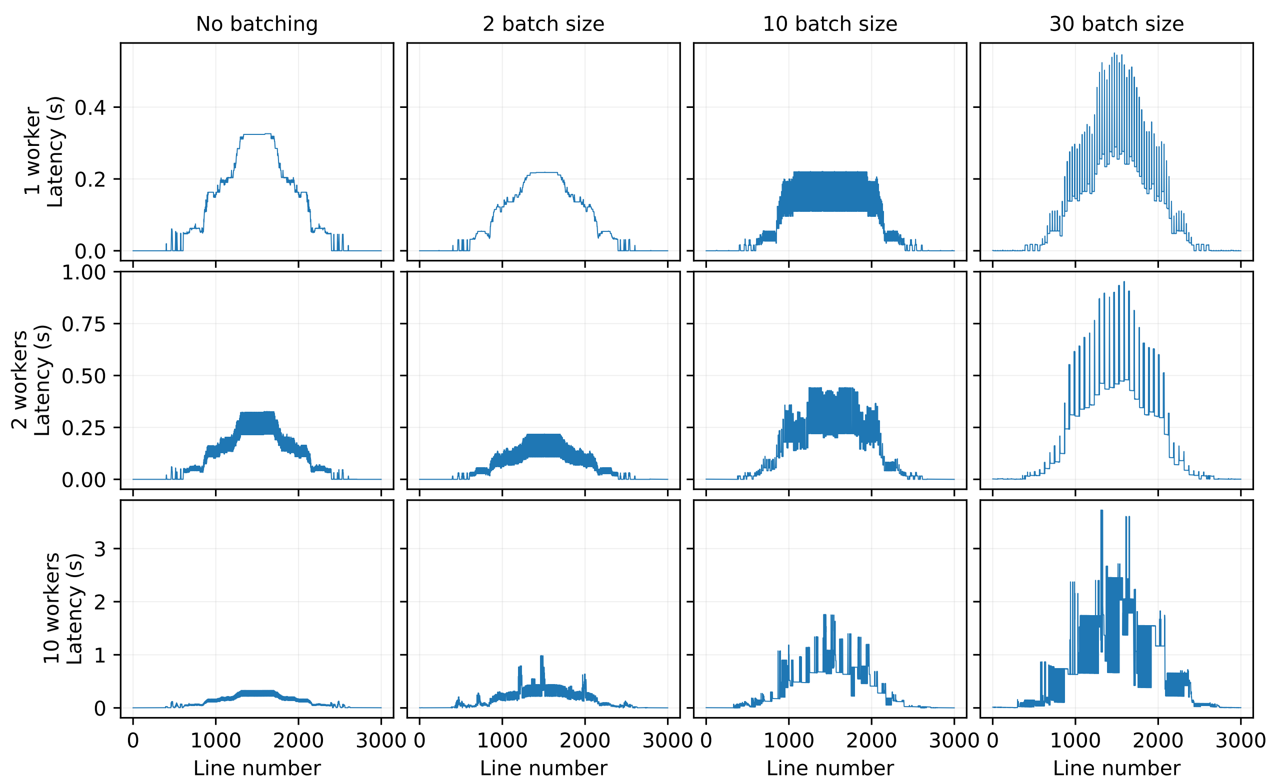


Figure 5.6: Latency profiling of different batch sizes in SPar (CUDA) Mandelbrot Streaming with the medium workload.

column of graphs in Figure 5.6 show that latencies of the CUDA-based version are an order of magnitude lower than the multi-core versions. In fact, all latencies are under 0.4 s for any number of workers: the highest latencies of these graphs are 326 ms for 1 worker, 327 ms for 2 workers, and 332 ms for 10 workers. Therefore, the use of the GPU highly improves the latencies for this application.

By definition, increasing the batch size increases the latency since the items wait in the worker of the replicated stage until it receives enough items to compute the batch. Nonetheless, the first and second graphs of the second column of Figure 5.6 (2 batch size) present lower latencies than the first column: the highest latencies are 219 ms for 1 worker and 218 ms for 2 workers. The reason is that the bottleneck is the GPU communication. Thus, the workers of the replicated stage must wait for the GPU response and the first stage waits for it to deliver the next item. This item is spending time waiting in the worker's queue. A batch size of 2 improves the latencies because the items are generated by the first stage faster than the replicated stages can process, so the worker of the replicated stage does not have to wait to receive the next item to compute the batch. In this case, the waiting time of the first item of each batch in the worker's batch vector is lower than the waiting time of the item in the worker's queue. This trend continues up to a batch size of 10 for 1 worker and up to a batch size of 5 for 2 workers, on which higher batch sizes start to degrade the latency times.

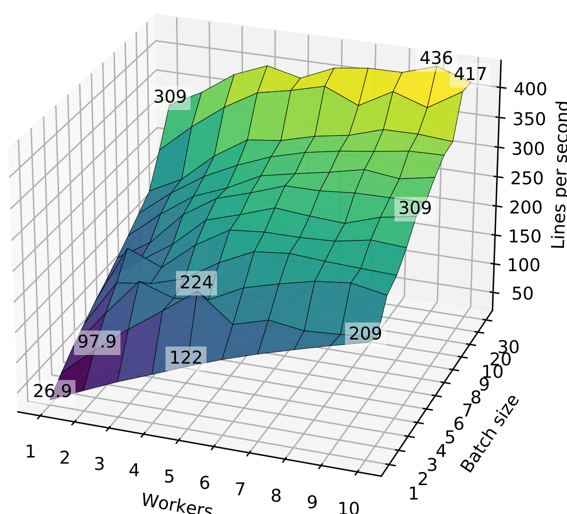
The last graph of the second column shows that the batch size of 2 degraded the performance when running with 10 workers, which is the highest latency of this test is 984 ms. This shows that the bottleneck of the application starts to shift towards the first stage as the number of parallel workers or batch sizes increases. The first stage is no longer capable of keep all 10 workers fed with items, thus some items wait longer in the batch vector of the replicated stage's worker until it receives the second item to compute the batch.

The last column of graphs in Figure 5.6, representing a batch size of 30 items, present a clear pattern of low and high latencies in short periods of time, among all number of workers. This can be seen more clearly in the second graph (2 workers with a batch size of 30) in the form of large plateaus of very stable latencies across a large number of lines. With such a high batch size, the first items that a worker receives remains a considerable amount of time in the batch vector until the vector is filled up and the batch is processed. These first items received by the stage represent the high latencies seen in the chart. Nonetheless, the last items of each batch are processed right after they are received by the worker. These items represents the lower latencies seen in the chart. Since the FastFlow scheduler (used by SPar) works in a round-robin fashion, increasing the number of workers also increases the amount of time necessary to fill the batch vector of each one. This is because the stage that invokes the GPU kernel is also responsible of building the batch of stream items. As an alternative, we could hand over to the first stage, the responsibility of building the batch and send this batch ready to be processed by the workers. However, this approach involves modifying the application flow and it may not be supported by all applications. Therefore, support for batching across the entire stream even without the GPU offloading (i.e.: decoupling the `Batch` and `Pure` attributes) is left as future work.

Many stream processing applications have strict requirements over latency, defined as service level objectives (SLO) thresholds [GVS+19]. Therefore, it is desirable to maximize the throughput while keeping the latency within acceptable levels [SRG+20]. Figure 5.7 presents the impact of different numbers of workers and batch sizes in the throughput (5.7a) and the maximum observed latency (5.7b) for the medium workload in the SPar (CUDA) version. We tested the application without batching (which is presented in Figure 5.7 as batch size of 1) and using batch sizes of 2 to 10, and also 20 and 30.

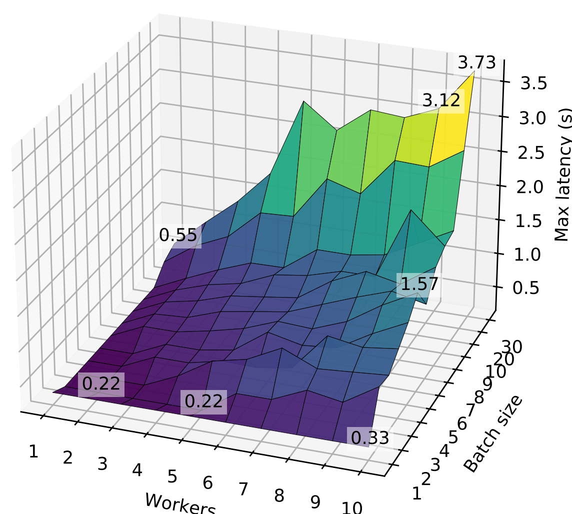
As previously discussed, for this workload, a batch size of 2 actually improves the latency. Therefore, the lowest maximum latency is 218 ms (0.22 s in Figure 5.7b), using 2 workers and a batch size of 2. Besides improving the latency, using a batch size of 2 offers a 86% of increase in the throughput: 97.9 lines/s using 2 workers and a batch size of 2 with respect to 52.5 lines/s using the same 2 workers but without batching. For this workload, the configuration of 5 workers with a batch size of 2 is also particularly interesting: it presents a throughput of 224 lines/s (84% more than the same configuration without batching, 122 lines/s) and the highest latency is only 220 ms (0.22 s in Figure 5.7b). Increasing the number of workers or the batch size impacts the latency more significantly because the items are

Mandelbrot with 3,000x3,000 fractal and 100k iterations



(a) Throughput of medium workload.

Mandelbrot with 3,000x3,000 fractal and 100k iterations



(b) Maximum latency of medium workload.

Figure 5.7: Effects of batching in SPar (CUDA) Mandelbrot Streaming for the medium workload.

waiting longer in the worker's batch vector. The best batch size depends on the workload characteristics, but also in the latency and throughput requirements of the application. Future works may offer automatic detection and adaptive batch sizes [SRG+20] based on higher-level SLO requirements [GVS+19].

5.6.2 Ray Tracing

Ray tracing is a technique for synthesizing illumination in 3-dimensional rendered scenes. It is based on a process called ray casting, which aims to find the closest object along the path of a ray. The basic idea of ray tracing is that, given a camera position (also referred as the eye) and a fixed screen size (or image plane), the computer can use the ray casting process to cast a ray for each pixel of the screen and calculates a color for this pixel that best represents the image in the scene behind it [Gla89]. Overall, this technique vaguely resembles the functioning of a classic pinhole camera.

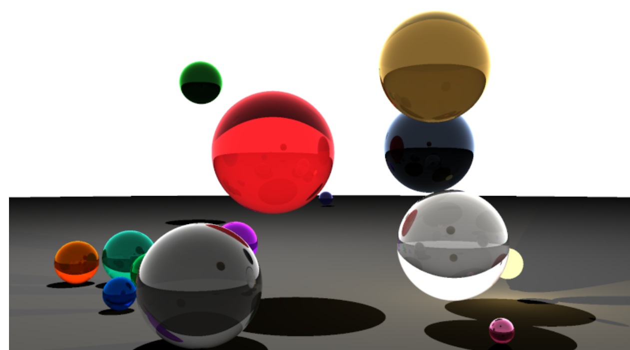
For each pixel, a ray (also known as view ray) is traced from the source camera to the direction of this specific pixel. Whenever the ray encounters an object, the object's surface properties such as surface color, emission of light (if the ray hit a light source), reflectivity, and transparency are taken into account to define the pixel color or to cast more rays. For example, if the ray encounters a mirror-like object (with high reflectivity), a reflection ray is cast from the point of intersection between the original ray and the object, towards the

reflection direction. Refraction rays works the same way as reflection rays, but with different directions. These refraction rays also weights differently for the final pixel colouring. Shadow rays are cast from the point where the ray hit the object towards each light source to check if there is any other object casting a shadow in this point or if it is directly illuminated by the light source [HAM19].

To further improve the photorealism of the generated image, a technique known as stochastic (or distributed) ray tracing casts a bunch of rays for each hit between the view ray and an object to check for light dispersion and diffusion. This technique widely improves the depiction of soft shadows and glossy surfaces [HAM19]. The ray tracing technique is largely used in the movies industry and more recently in the game industry to create realistic computer-generated (CGI) scenes.

Real-time ray tracing is considered the holy grail of computer graphics [Bel20]. Even though classic ray tracing algorithm is embarrassingly parallel [HAM19] and the technique is largely known [App68], it was only the recent GPU hardware improvements that opened up the possibility that ray tracing could be joining the set of real-time applications. Figure 5.8 depicts two examples of images from ray tracing applications: the left one (5.8a) is a scene commonly used as benchmark in ray tracing applications, while the right one (5.8b) is a single frame from the output of the ray tracing application demonstrated in Listing 5.3.

(a) Amazon Lumberyard Bistro scene commonly used as benchmark for ray tracing applications [Ama17].



(b) An example of output frame from our ray tracing application.

Figure 5.8: Sample images of ray tracing applications.

Our ray tracing application generates a video as a stream of frames depicting a CGI scene purely composed of a pre-defined number of animated spheres. An output frame example composed of 16 spheres is presented in Figure 5.8b (the gray floor and the white sky are also just very big spheres). Listing 5.3 present the streaming region of this application. It exemplifies the application of the Transformation Rule 5.2 (however, here the `Pure` attribute is used as ID) which generates the Map pattern nested inside the replicated stage of the Farm pattern.

```
struct tVec3f { float x; float y; float z; };
typedef struct tVec3f Vec3f;
struct tSphere {
  Vec3f center;
  float radius;
  Vec3f surfaceColor, emissionColor;
  float transparency, reflection;
  int animation_frame;
  Vec3f animation_position;
};
typedef struct tSphere Sphere;
void raytrace(int totalFrames, int width, int height, std::vector<Sphere>
    initialSpheres) {
  float invWidth = 1 / float(width);
  float invHeight = 1 / float(height);
  float fov = 30;
  float aspectRatio = width / float(height);
  float angle = tan(M_PI * 0.5 * fov / 180.);
  int imgsize = width * height;
  [[spar::ToStream, spar::Input(totalFrames, width, height, imgsize,
    initialSpheres, invWidth, invHeight, aspectRatio, angle)]]
  for (int frame = 1; frame <= totalFrames; frame++) {
    int spheres_size = initialSpheres.size();
    Sphere* spheres = new Sphere[spheres_size];
    memcpy(spheres, initialSpheres.data(), sizeof(Sphere) * spheres_size);
    for(unsigned long i = 0; i != spheres_size; i++) {
      computeSpheresPosition(spheres, spheres_size);
    }
    Vec3f *img;
    [[spar::Stage, spar::Input(frame, width, height, imgsize, spheres,
    spheres_size, invWidth, invHeight, aspectRatio, angle, img),
    spar::Output(img), spar::Replicate()]] {
      img = new Vec3f[imgsize];
      [[spar::Pure, spar::Input(frame, width, height, spheres[spheres_size],
    spheres_size, invWidth, invHeight, aspectRatio, angle),
    spar::Output(img[imgsize])]]
      for (unsigned y = 0; y < height; ++y) {
        for (unsigned x = 0; x < width; ++x) {
          float xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectRatio;
          float yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
          Vec3f raydir;
          raydir.x = xx;
          raydir.y = yy;
          raydir.z = -1;
          normalize(&raydir);
          Vec3f rayorig;
          rayorig.x = 0;
          rayorig.y = 0;
          rayorig.z = 0;
          img[y*width+x] = trace(&rayorig, &raydir, spheres, spheres_size, 0);
        }
      }
    }
    [[spar::Stage, spar::Input(img)]] {
      show(img);
    }
  }
}
```

Listing 5.3: Ray tracing annotated with SPar using the new Pure attribute.

A structure containing three floating-point numbers is defined in line 1. We use this structure to represent three-dimensional points in space and also to represent RGB colors. For example, each pixel of the image is represented by an instance of this structure as well as the surface and emission colors of the spheres. Each sphere in the scene is represented by the structure defined in line 3. It contains: the initial position of the sphere's center in the three-dimensional scene, the radius, the surface color, the emission color (if the sphere is a light source), the degrees of transparency and reflection, and the definition of the animation that should be applied to the sphere in each frame.

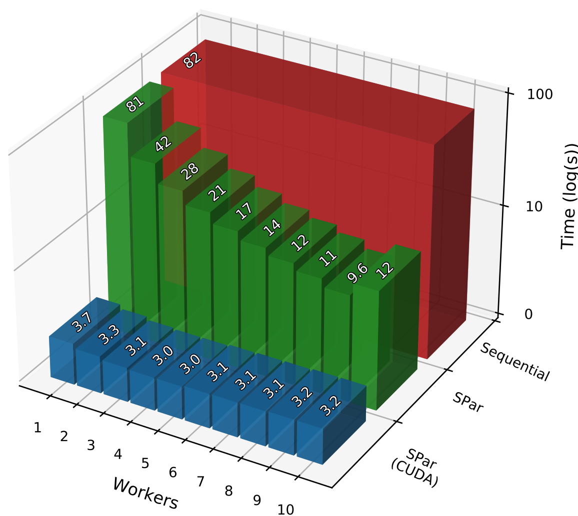
The main `raytrace` function is defined in line 12. It receives as arguments the number of frames that will be generated, the width and height of the frames, as well as the initial configuration for the spheres. After preparing some auxiliary variables, the streaming region is delimited by the `ToStream` annotation in line 19. The first part of the stream processing loop copies the spheres' initial information (line 23) and updates the position of them based on the current frame number (line 25). The replicated `Stage` in line 28 is responsible for casting a ray for each pixel of the scene to calculate the pixel color based on the spheres and their properties. Since the calculation for each pixel is independent, we can annotate this part of the code with the `Pure` attribute in line 30. For the sake of simplicity, we did not include the declaration of the recursive `trace` function called in line 44, which is responsible for tracing the rays needed to determine the pixel color. All the computation occurs in the computer memory and the last `Stage` (line 48) is responsible for showing the image (line 49 in Listing 5.3) or saving the image in the persistent storage. We removed this step during the time measurement to avoid incurring I/O times in the performance tests.

We tested three workloads for the ray tracing application, with frames of 640x360, 1280x720, and 1920x1080 pixels of resolution. We used the same scene (depicted in Figure 5.8b) for all workloads, which are composed of 16 spheres, and the same number of frames: 1,200. The graphs in Figures 5.9, 5.10, and 5.11 presents the performance results for the three workloads, respectively. The throughput of this ray tracing application is defined as the number of frames processed per second (FPS), since each stream item is a single frame of the output video. Given that `Batch` can only be used together with `Stage` and `Pure`, and the `Pure` is used as an ID attribute in this application, we did not include the `Batch` attribute in this tests. Moreover, each stream item already presents enough workload for the GPU kernel, thus it is not necessary to change the stream granularity.

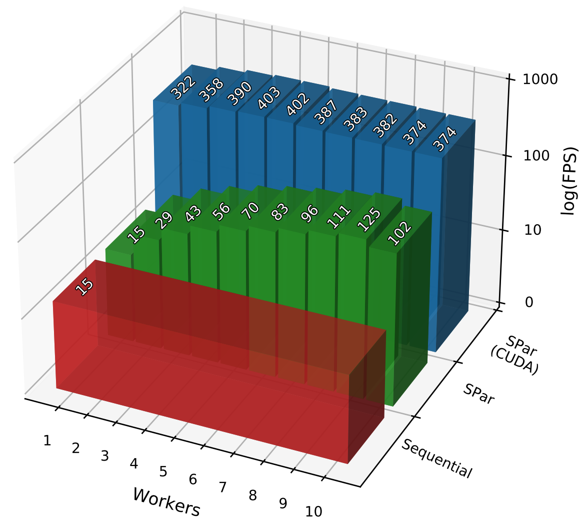
For this application we were not able to test the OpenCL backend. The `trace` function is recursive and receives the `spheres` parameter, which is a pointer of memory in global address space. In this situation, the code generated by the SPar compiler is not compatible with OpenCL 1.2 because it requires the function arguments to be annotated with `__global` in order to receive pointers to global memory addresses, which is not valid C++. To overcome this limitation, OpenCL 2 introduces the generic address space [Khr18, LI15], which provides automatic detection of memory address spaces. Unfortunately, as discussed

in Section 4.8, the NVIDIA OpenCL driver only offers support for OpenCL version 1.2 [Khr20a]. Since we only have access to devices of this manufacturer, we were not able to evaluate this application using the OpenCL backend.

Figure 5.9 presents the execution time (5.9a) and throughput (5.9b) for the small workload. The multi-core version presents an increasing performance up to nine parallel workers of the replicated stage, with a top performance of 9.6 seconds (8.6$\times$ speedup with respect to the sequential version) or 125 FPS. Nonetheless, the GPU version presents a substantial performance improvement compared to both the multi-core and sequential versions, even when using a single worker, which takes 3.7 seconds (22.1$\times$ speedup with respect to the sequential version) to run and present a frame rate of 322 FPS. This difference is expected since the application is specially suited for GPU parallelism. The top performance is presented by the GPU version using four parallel workers, with an execution time of 2.98 seconds and a frame rate of 403 FPS. Table 5.5 shows that the only test that presented a SEM above 100 ms was the sequential version, with an standard error of 118 ms.



(a) Execution time of small workload.



(b) Throughput of small workload.

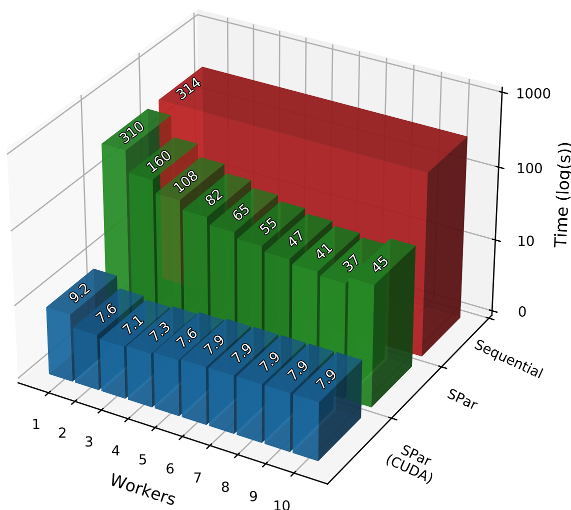
Figure 5.9: Performance results of the ray tracing application for the small workload.

Table 5.5: SEM in seconds of tests for Figure 5.9a.

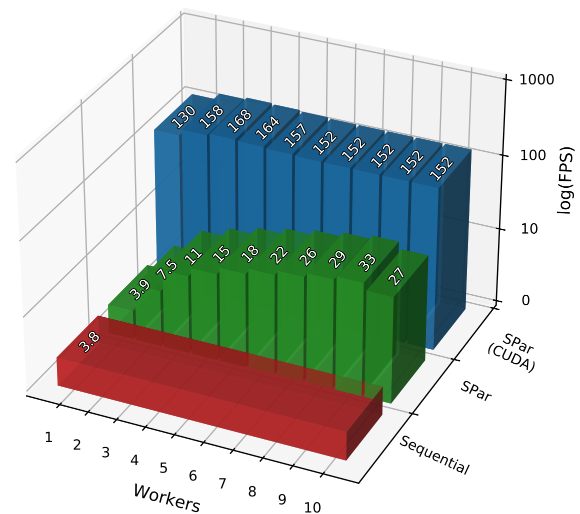
Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.011	0.017	0.012	0.010	0.009	0.017	0.015	0.014	0.009	0.011
SPar	0.022	0.016	0.005	0.006	0.004	0.092	0.084	0.023	0.003	0.004
Sequential	0.118									

Figure 5.10 presents the execution time (5.10a) and throughput (5.10b) for the medium workload. The behavior of the application in this workload is very similar to the small

workload, with the multi-core version obtaining its highest performance using nine parallel workers (execution time of 37 seconds, 8.6× speedup with respect to the sequential version, and 32.8 FPS). The highest performance was achieved by the GPU version using three parallel workers (execution time of 7.13 seconds, 44× speedup with respect to the sequential version, and 168 FPS). The difference of the GPU version compared to the CPU versions is bigger than the small workload, which is explained by the fact that the stream item presents a heavier load and therefore is more suitable to GPU parallelism. The SEM of these tests are presented in Table 5.6. Just like the small workload, the multi-core and GPU versions presented SEM below 100 ms for all configurations of parallel workers and the sequential version presented an standard error of 164 ms.



(a) Execution time of medium workload.



(b) Throughput of medium workload.

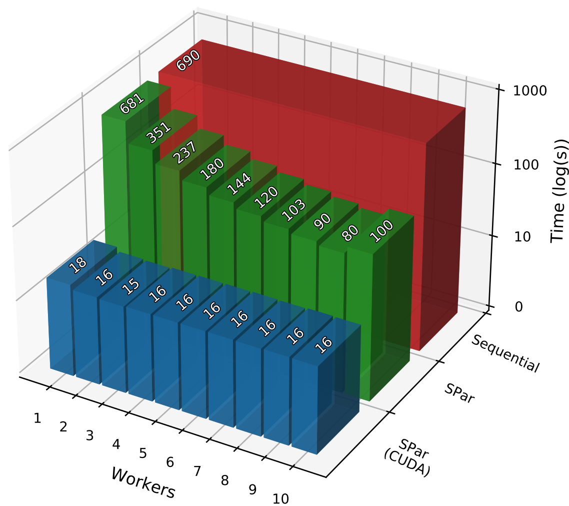
Figure 5.10: Performance results of the ray tracing application for the medium workload.

Table 5.6: SEM in seconds of tests for Figure 5.10a.

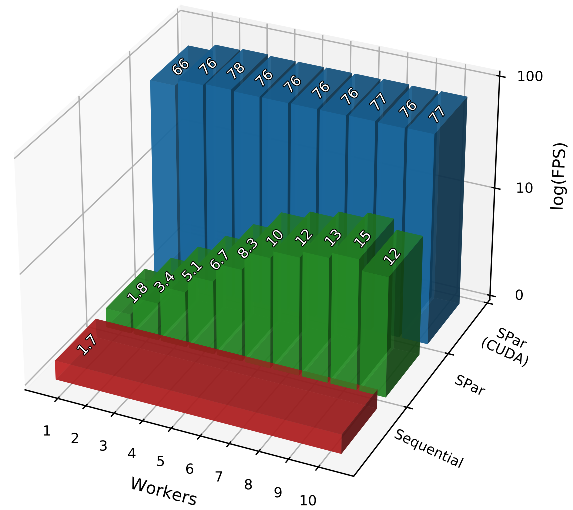
Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.009	0.008	0.020	0.014	0.012	0.018	0.012	0.012	0.009	0.013
SPar	0.065	0.036	0.022	0.013	0.016	0.055	0.016	0.011	0.026	0.008
Sequential	0.164									

Figure 5.11 presents the execution time (5.11a) and throughput (5.11b) for the large workload. Similar to the previous workloads, the highest performance for the multi-core version was presented with nine workers (execution time of 80.3 seconds, 8.6× speedup with respect to the sequential version, and 14.9 FPS) while the highest performance for the GPU version was presented with three workers (execution time of 15.4 seconds, 44.8× speedup with respect to the sequential version, and 77.9 FPS). It worth noting that the speedup of

the GPU version with respect to the sequential version increased together with the increase of the workload of each stream item while the CPU version presented a speedup of 8.6× for all workloads. This highlights that offloading the computations to the GPU in stream processing applications is specially interesting when the stream item requires a reasonable amount of work. If the stream item does not present enough computation intensity to worth offloading to the GPU, the `Batch` attribute can be used to increase the amount of work of each stream item. The SEM of these tests are presented in Table 5.6. Just like the small and medium workloads, the multi-core and GPU versions presented SEM below 100 ms for all configurations of parallel workers while the sequential version presented an standard error of 146 ms.



(a) Execution time of large workload.



(b) Throughput of large workload.

Figure 5.11: Performance results of the ray tracing application for the large workload.

Table 5.7: SEM in seconds of tests for Figure 5.11a.

Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.038	0.023	0.030	0.022	0.029	0.028	0.022	0.025	0.030	0.017
SPar	0.078	0.254	0.032	0.027	0.025	0.018	0.008	0.019	0.013	0.015
Sequential	0.146									

Note that the multi-core version is unable to present real-time frame rate on the large workload (which generates images in Full HD resolution), peaking at 15 FPS, even when considering the lowest acceptable real-time frame rate threshold of 25 FPS. On the other hand, the GPU version is able to sustain real-time frame rate on this workloads, running at 77.9 FPS, which is higher than the industry standard of real-time frame rate threshold of 60 FPS [HAM19].

In summary, we observed that the ray tracing application exposes massive data parallelism and obtain higher throughput numbers using the GPU to process the images. We did not use the `Batch` attribute in this application because the stream item granularity is already suitable for massive parallelism exploitation. In fact, even in the small workload the image that represents the stream item is big enough to worth offloading the computation to the accelerator device. We also observed that increasing the number of parallel multi-core workers did not result in increased performance when using the GPU to process the images because the GPU becomes the bottleneck of the application. All tests presented low SEM, which indicates our results are reliable. We show that by using the novel `Pure` attribute the programmer is able to exploit the computational power of the GPU by adding a single annotation while obtaining speedups of 5.2× with respect to the best performance exploiting multi-core CPU parallelism. The ray tracing application also demonstrates that SPar is able to automatically generate GPU kernels from C++ sequential code. Also, it challenges SPar to deal with a complex code structure where there are nested custom structures (such as the `Vec3f` and `Sphere` structures) and recursive function calls (such as the `trace` function).

5.6.3 Lane Detection

Market analysts have been predicting a boom of self-driving vehicles in following years [ABI18, Gar19]. This is a hot area of research and development with major investments both from industry and academia [End17]. However, detecting the road lanes is one of the first major challenges for developing a Driving Automation System (DAS) [HLLR14, SAE18]. While Light Detection and Ranging (LiDAR) sensors are used to detect other cars and objects surrounding the DAS [KP08], detecting the lanes of the road depends on camera images and sophisticated algorithms for edge detection. One of such algorithms is the combination of the Canny edge detector [Can86] and the Hough transform [Bal81].

Listing 5.4 presents our version of the lane detection application using this combination [WTS04]. We adapted an OpenCV[1] based implementation[2] to express stream parallelism. The streaming region is delimited by the `ToStream` annotation in line 2. The first stage that is defined in lines 4–7. It reads the next image frame (which could come from a camera, but in our application it comes from static images previously loaded from the persistent storage to the computer memory), prepares the vertices of the region of interest (ROI) and sends them to the next stage. The replicated `Stage` in line 8 represents the workers, which performs the detection of the lane in the input image and sends the detected lines to the next `Stage`. The final `Stage` applies the detected lanes as red lines in the image using OpenCV's `line` function and shows the output image on screen using OpenCV's `imshow` function.

[1] https://opencv.org/
[2] https://github.com/Sujay-k/lane-detection-using-cpp

```
bool lanedetection() {
  [[spar::ToStream]]
  while (true) {
    cv::Mat imgContent = getNextImage();
    if (!imgContent) break;
    std::vector<cv::Point*> vertices = getRoiVertices();
    std::vector<cv::Vec4i> lines;
    [[spar::Stage, spar::Input(imgContent, vertices, lines),
   spar::Output(lines), spar::Replicate()]] {
      int rows = imgContent.rows;
      int cols = imgContent.cols;
      int imgsize = rows * cols;
      unsigned char *imgData = imgContent.data;
      // Gauss filter
      int gaussFilterSize = 5*5;
      double *gaussFilter = getGaussFilter(gaussFilterSize); // Builds the
   normalized 5x5 matrix
      unsigned char *blurredImg = new unsigned char[imgsize];
      // Fill the borders where we can't apply the filter
      for (int i = 0; i < 2; i++) {
        for (int j = 0; j < cols; j++) {
          if (j < 2 || j >= cols-2) {
            for (int ii = 0; ii < rows; ii++) { //Avoids an extra loop
              blurredImg[ii*cols + j] = imgData[ii*cols + j];
            }
          }
          blurredImg[i*cols + j] = imgData[i*cols + j];
          blurredImg[(rows-1-i) * cols + j] = imgData[(rows-1-i)*cols + j];
        }
      }
      [[spar::Pure, spar::Input(rows, cols, gaussFilter[gaussFilterSize],
   imgData[imgsize], blurredImg[imgsize]), spar::Output(blurredImg[imgsize])]]
      for (int i = 2; i < rows - 2; i++) {
        for (int j = 2; j < cols - 2; j++) {
          double sum = 0;
          for (int x = 0; x < 5; x++) {
            for (int y = 0; y < 5; y++) {
              sum += gaussFilter[x*5+y] * (double)(imgData[(i + x - 2)*cols +
   (j + y - 2)]);
            }
          }
          blurredImg[i*cols+j] = sum;
        }
      }
      // Sobel filter
      unsigned char *sobelImg;
      int sobelFilterSize = 3*3;
      double *xFilterSobel = getSobelFilter(sobelFilterSize, 'x');
      double *yFilterSobel = getSobelFilter(sobelFilterSize, 'y');
      float *anglesMap = new float[imgsize];
      // Fill the borders where we can't apply the filter
      for (int i = 0; i < 1; i++) {
        for (int j = 0; j < cols; j++) {
          if (j < 1 || j >= cols-1) {
            for (int ii = 0; ii < rows; ii++) { //Avoids an extra loop
              sobelImg[ii*cols + j] = 0;
            }
          }
          sobelImg[i*cols + j] = 0;
```

```
56          sobelImg[(rows-1-i)*cols + j] = 0;
57          anglesMap[i*cols + j] = 0;
58          anglesMap[(rows-1-i)*cols + j] = 0;
59        }
60      }
61      [[spar::Pure, spar::Input(rows, cols, xFilterSobel[sobelFilterSize],
    yFilterSobel[sobelFilterSize], blurredImg[imgsize], anglesMap[imgsize],
    sobelImg[imgsize]), spar::Output(anglesMap[imgsize], sobelImg[imgsize])]]
62      for (int i = 1; i < rows - 1; i++) {
63        for (int j = 1; j < cols - 1; j++) {
64          double sumx = 0;
65          double sumy = 0;
66          for (int x = 0; x < 3; x++) {
67            for (int y = 0; y < 3; y++) {
68              sumx += xFilterSobel[x*3 + y] * (double)(blurredImg[(i + x -
    1)*cols + j + y - 1]);
69              sumy += yFilterSobel[x*3 + y] * (double)(blurredImg[(i + x -
    1)*cols + j + y - 1]);
70            }
71          }
72          double sumxsq = sumx*sumx;
73          double sumysq = sumy*sumy;
74          double sq2 = sqrt(sumxsq + sumysq);
75          if(sq2 > 255) { //Unsigned Char Fix
76            sq2 =255;
77          }
78          sobelImg[i*cols+j] = sq2;
79          if(sumx == 0) { //Arctan Fix
80            anglesMap[i*cols + j] = 90;
81          } else {
82            anglesMap[i*cols + j] = atan(sumy/sumx);
83          }
84        }
85      }
86      // Non-maximum suppression
87      unsigned char *edgesImg = new unsigned char[imgsize];
88      // Fill the borders where we can't apply the filter
89      for (int i = 0; i < 1; i++) {
90        for (int j = 0; j < cols; j++) {
91          if (j < 1 || j >= cols-1) {
92            for (int ii = 0; ii < rows; ii++) { //Avoids an extra loop
93              edgesImg[ii*cols + j] = 0;
94            }
95          }
96          edgesImg[i*cols + j] = 0;
97          edgesImg[(rows-1-i)*cols + j] = 0;
98        }
99      }
100     [[spar::Pure, spar::Input(rows, cols, anglesMap[imgsize],
    sobelImg[imgsize], edgesImg[imgsize]), spar::Output(edgesImg[imgsize])]]
101     for (int i = 1; i < rows-1; i++) {
102       for (int j = 1; j < cols-1; j++) {
103         float Tangent = anglesMap[i*cols+j];
104         edgesImg[i*cols+j] = sobelImg[i*cols+j];
105         //Horizontal edge
106         if ((-22.5 < Tangent && Tangent <= 22.5) || (157.5 < Tangent &&
    Tangent <= -157.5)) {
107           if (sobelImg[i*cols+j] < sobelImg[i*cols + j+1] ||
    sobelImg[i*cols+j] < sobelImg[i*cols + j-1]) edgesImg[i*cols+j] = 0;
```

```
108            }
109            //Vertical edge
110            if ((-112.5 < Tangent && Tangent <= -67.5) || (67.5 < Tangent &&
       Tangent <= 112.5)) {
111              if (sobelImg[i*cols+j] < sobelImg[(i+1)*cols + j] ||
       sobelImg[i*cols+j] < sobelImg[(i-1)*cols + j]) edgesImg[i*cols+j] = 0;
112            }
113            //-45 Degree edge
114            if ((-67.5 < Tangent && Tangent <= -22.5) || (112.5 < Tangent &&
       Tangent <= 157.5)) {
115              if (sobelImg[i*cols+j] < sobelImg[(i-1)*cols + j+1] ||
       sobelImg[i*cols+j] < sobelImg[(i+1)*cols + j-1]) edgesImg[i*cols+j] = 0;
116            }
117            //45 Degree edge
118            if ((-157.5 < Tangent && Tangent <= -112.5) || (22.5 < Tangent &&
       Tangent <= 67.5)) {
119              if (sobelImg[i*cols+j] < sobelImg[(i+1)*cols + j+1] ||
       sobelImg[i*cols+j] < sobelImg[(i-1)*cols + j-1]) edgesImg[i*cols+j] = 0;
120            }
121          }
122        }
123        // Thresholds
124        for (int i = 1; i < rows-1; i++) {
125          for (int j = 1; j < cols-1; j++) {
126            if (edgesImg[i*cols + j] > 150) {
127              edgesImg[i*cols + j] = 255;
128            } else if(edgesImg[i*cols + j] < 50) {
129              edgesImg[i*cols + j] = 0;
130            } else {
131              bool anyHigh = false;
132              bool anyBetween = false;
133              for (int x = i-1; !anyHigh && x < i+2; x++) {
134                for (int y = j-1; !anyHigh && y < j+2; y++) {
135                  if (x >= 0 && y >= 0 && x < imgrows && y < imgcols) {
136                    if (edgesImgData[x*imgcols + y] > 150) {
137                      edgesImgData[i*imgcols + j] = 255;
138                      anyHigh = true;
139                    } else if(edgesImgData[x*imgcols + y] >= 50) {
140                      anyBetween = true;
141                    }
142                  }
143                }
144              }
145              if (!anyHigh && anyBetween) {
146                for (int x = i-2; !anyHigh && x < i+3; x++) {
147                  for (int y = j-1; !anyHigh && y<j+3; y++) {
148                    if (x >= 0 && y >= 0 && x < imgrows && y < imgcols) {
149                      if (edgesImgData[x*imgcols + y] > 150) {
150                        edgesImgData[i*imgcols + j] = 255;
151                        anyHigh = true;
152                      }
153                    }
154                  }
155                }
156              }
157              if(!anyHigh) edgesImg[i*cols + j] = 0;
158            }
159          }
160        }
```

```
161       Mat edges = Mat(rows, cols, CV_8UC1, edgesImg);
162       Mat roi = Mat::zeros(imgContent.size(),imgContent.type());
163       int numberOfpoints = vertices.size()
164       cv::fillPoly(roi, vertices.data(), &numberOfpoints, 1, Scalar(255), 8);
165       bitwise_and(edges, roi, roi);
166       cv::HoughLines(roi, lines, 1, CV_PI/360, 40, 120, 280);
167       cvtColor(imgContent, imgContent, COLOR_GRAY2BGR);
168       delete[] gaussFilter;
169       delete[] blurredImg;
170       delete[] sobelImg;
171       delete[] xFilterSobel;
172       delete[] yFilterSobel;
173       delete[] anglesMap;
174       delete[] edgesImg;
175     }
176     [[spar::Stage, spar::Input(imgContent, lines)]] {
177       for (auto l=0; l<lines.size(); l++) {
178         Vec4i ln = lines[l];
179         cv::line(imgContent, Point(ln[0],ln[1]), Point(ln[2],ln[3]),
    Scalar(0,0,255), 1, LINE_AA);
180       }
181       cv::imshow("Detected lanes", imgContent);
182     }
183   }
184 }
```

Listing 5.4: Lane detection annotated with SPar using the new Pure attribute.

In order to exploit the GPU parallelism with SPar we broke down the Canny algorithm in its components[3,4]. Thus, instead of using the `cv::Canny` function we apply a set of filters to the input image: (a) noise reduction, through the application of a Gaussian filter (lines 13–40); (b) find the intensity gradient using the Sobel filter as edge detection operator (lines 41–85). Based on this edge detection we also check the direction of each pixel (line 79); (c) non-maximum suppression to thin the detected edges based on four discrete directions (lines 105–120); and (d) double hysteresis thresholds to remove the weak edges and keep only the strong edges (lines 123–160). In Listing 5.4, the lower and upper thresholds are fixed as 50 and 150. With the exception of the last one (thresholds), which carries a dependency over previous iterations, each of these steps is annotated with `Pure` in Listing 5.4. After applying these filters, any edges outside the predefined ROI are discarded (lines 162–165) and the Hough transform algorithm is applied to the detected edges (line 166) using the OpenCV's `HoughLines` function.

Figure 5.12 presents an output example of each step of this application. The first image is the original input, which is read in grayscale using the `cv::imread` function with the `IMREAD_GRAYSCALE` flag. The second image presents the `blurredImg` variable contents after the application of the Gaussian blur to the black and white image (line 40 of Listing 5.4). The third image presents the `sobelImg` variable contents after the application of the Sobel filter (line 85). The fourth image presents the `edgesImg` variable contents in line 160, after the

[3] https://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html
[4] https://github.com/hasanakg/Canny-Edge-Detector

application of the non-maximum suppression and the thresholds. The fifth image presents the `roi` variable in line 165, which contains only the edges inside the ROI. The sixth and last image presents the output image (`imgContent` in line 181) after applying the Hough transform algorithm [Bal81] and drawing red lines to delimit the detected lanes.

Figure 5.12: Lane detection steps output examples for frame 3 of KITTI 027 dataset.

In Listing 5.4 we chose to keep all four steps of Canny algorithm in the same `Stage` and annotate the loops with `Pure`. However, if one of such steps is a bottleneck, the application programmer could easily define a separate `Stage` for each step and use a different parallelism degree for each one. This design would also allow for the `Batch` attribute to be used, since

the Pure would be used as AUX attribute together with Stage. Therefore, the Batch could be used in the stage before the heavier one to balance the workload of each Stage and provide performance improvements. The high-level of abstraction provided by SPar allows the programmer to quickly prototype version candidates by changing the attributes and testing which version better fits their needs. It is up to the programmer to decide which loops should be offloaded to GPU and how to parallelize the application's execution flow.

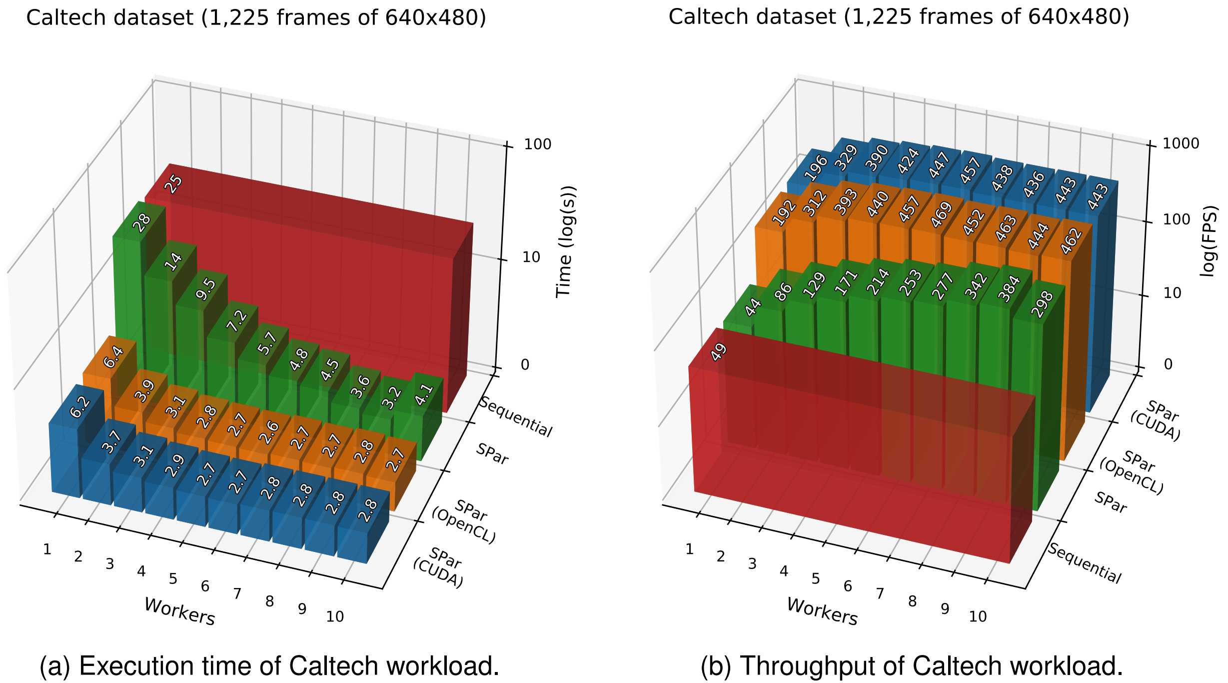
There are plenty of datasets focused in tuning lane detection algorithms which aims in the comparison of the accuracy of different lane detection solutions [SU19]. Since our focus is solely in improving the algorithm performance through the exploitation of stream and data parallelism, we did not calculate the application's accuracy for the detected lanes. However, we did use two of these datasets for testing our lane detection application: (a) the four sequences (or clips) that compose the Caltech Lanes dataset [Aly08], namely cordova1, cordova2, washington1, and washington2. This set contains 1,225 frames with 640x480 resolution, which amounts to 535 MiB of data in PNG format; (b) eight sequences of the KITTI dataset [GLSU13] categorized in the "Road" category which were captured in September 26th 2011, numbered as: 015, 027, 028, 029, 032, 052, 070, and 101. This subset contains 3,169 frames with 1242x375 resolution, which amounts to 809 MiB of data in PNG format.

Figure 5.13 presents the execution time (5.13a) and throughput (5.13b) for the Caltech dataset. SPar's multi-core version presented the best performance numbers with nine workers, processing the 1,225 frames in 3.19 seconds (7.9× speedup with respect to the sequential version and 384 FPS). The SPar version with GSPARLIB's CUDA backend did not present any significant differences with more than five workers with execution times ranging from 2.68 seconds (with six workers) to 2.82 seconds (with eight workers) and frame rates between 457 and 436 FPS. With respect to the sequential version, these times represent 9.4× and 8.9× speedup, respectively. The SPar version with GSPARLIB's OpenCL backend presented stable performance using any number of workers between five and ten with an average execution time of 2.69 seconds (±95 milliseconds), which represents 9.4× speedup with respect to the sequential version and a frame rate of 458 FPS. The SEM of all tests are shown in Table 5.8 and are below 100 ms.

Table 5.8: SEM in seconds of tests for Figure 5.13a.

Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.014	0.047	0.029	0.042	0.023	0.011	0.071	0.043	0.124	0.015
SPar (OpenCL)	0.019	0.041	0.004	0.051	0.021	0.029	0.072	0.064	0.082	0.051
SPar	0.035	0.059	0.055	0.037	0.026	0.028	0.207	0.003	0.002	0.078
Sequential	0.002									

Figure 5.14 presents the execution time (5.14a) and throughput (5.14b) for the KITTI dataset. SPar's multi-core version presented performance improvements up to nine parallel workers, which processed the 3,169 frames in 12.4 seconds (7.9× speedup with respect to



(a) Execution time of Caltech workload.

(b) Throughput of Caltech workload.

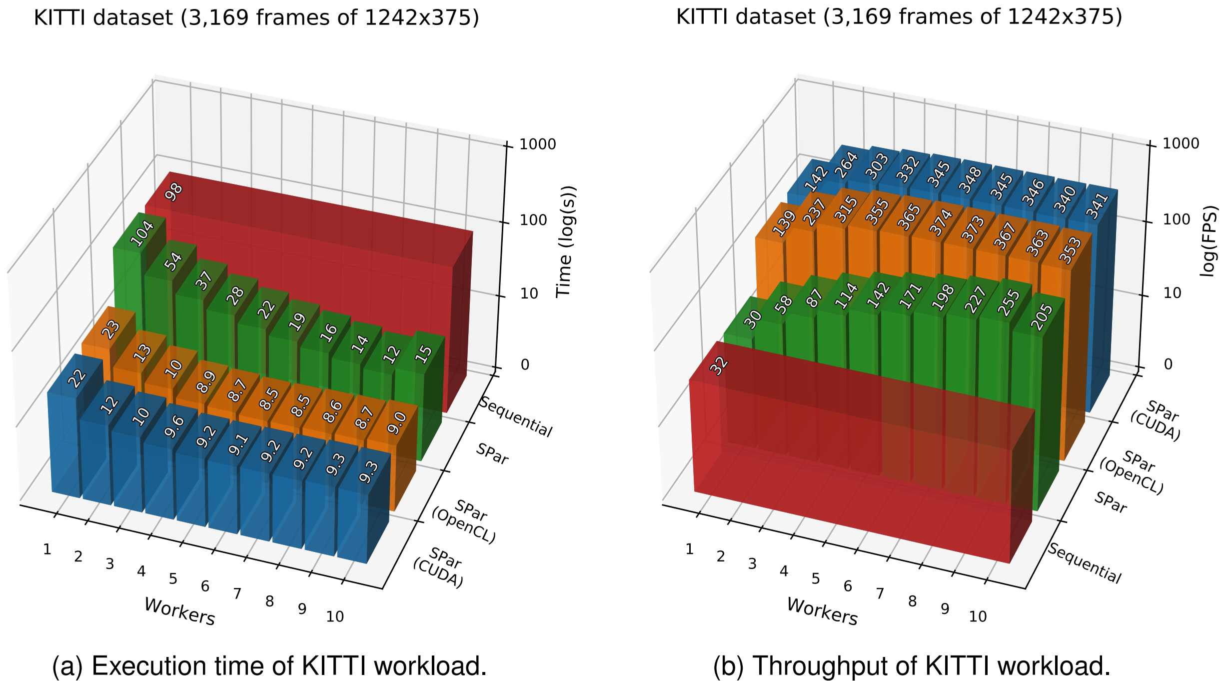
Figure 5.13: Performance results of lane detection for the Caltech dataset.

the sequential version and 255 FPS). The SPar version with GSPARLIB's CUDA backend presented the best performance using six workers, with an execution time of 9.12 seconds (10.7× speedup with respect to the sequential version and 348 FPS). The SPar version with GSPARLIB's OpenCL backend presented the best performance using six workers, with an execution time of 8.48 seconds (11.5× speedup with respect to the sequential version and 374 FPS). Interestingly, in this application the OpenCL version surpassed the CUDA version's performance. Table 5.9 presents the standard errors for these tests. All the GPU versions as well as the sequential version and the multi-core versions with more than one worker presented SEM below 100 ms. However, the multi-core version presented a standard error of 435 ms using a single worker thread.

Table 5.9: SEM in seconds of tests for Figure 5.14a.

Name	*Workers*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
SPar (CUDA)	0.085	0.122	0.178	0.119	0.089	0.081	0.077	0.082	0.086	0.054
SPar (OpenCL)	0.082	0.089	0.034	0.011	0.060	0.050	0.054	0.049	0.033	0.081
SPar	0.435	0.015	0.030	0.072	0.046	0.005	0.052	0.005	0.004	0.021
Sequential	0.011									

The three pure regions of the lane detection application represent the image filters applied in the input image. These regions are simpler than the pure region of the ray tracing application, which was challenging for automatic GPU offloading because it uses complex code inside the GPU kernel, as discussed in Section 5.6.2. However, the pure loops of the lane detection application are interleaved with sequential blocks, which requires complex data



(a) Execution time of KITTI workload.

(b) Throughput of KITTI workload.

Figure 5.14: Performance results of lane detection for the KITTI dataset.

management and separate GPU kernel invocations. Thus, it presents a different challenge to the SPar compiler. By using our novel transformation rules presented in Section 5.4, SPar compiler is able to correctly generate GPU parallel code and achieve sequential equivalence.

We were able to use both CUDA and OpenCL backends to exploit the GPU parallelism in this application. Overall, the performance of the OpenCL version was better than the CUDA counterpart. This result surprised us, since it is the opposite of what we found out when testing the Mandelbrot Streaming application in the same environment, as discussed in Section 5.6.1. Nonetheless, we observed that both GPU versions presented better performance than the multi-core version. Similarly to the ray tracing application, we also observed that increasing the number of parallel multi-core workers did not result in increased performance when using the GPU to process the images because the GPU becomes the bottleneck of the application. Thanks to the SPar high-level API, the programmer is able to exploit the GPU parallelism by adding a single annotation in each of the pure regions of the code.

5.7 Performance Difference of Code Generation and Manual Implementation

In this section we discuss the performance difference between the code generated by our implementation in the SPar compiler and handwritten code using FastFlow and GSPARLIB for combined stream and data parallelism. The underlying tools and runtimes as well as the approach for parallelizing the applications are the same. Therefore, we are

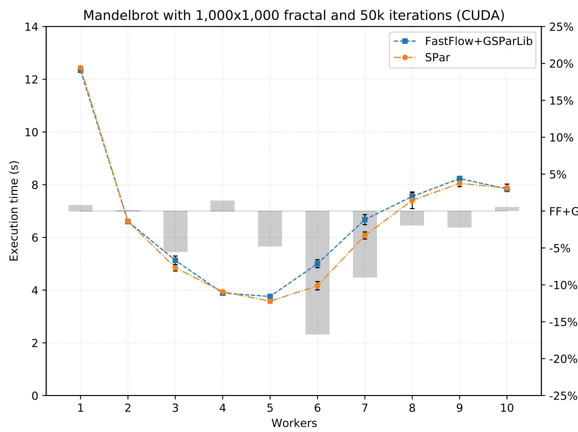
measuring only the impact of handwritten parallel code versus the automatic parallel code generation, using the total execution time of the applications as the performance metric.

5.7.1 Mandelbrot Streaming

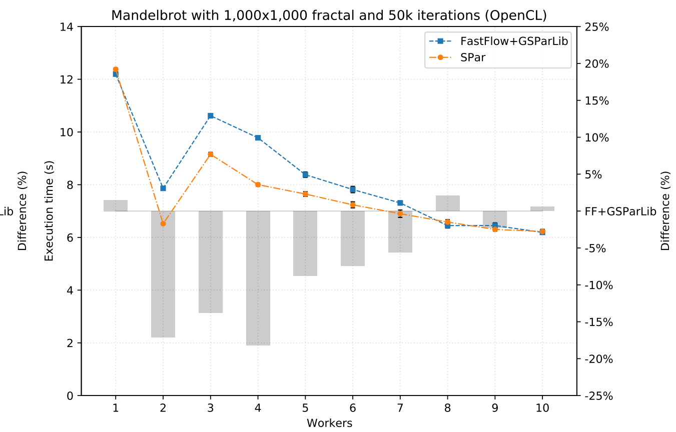
Figure 5.15 presents a performance comparison between automatic code generation (SPar) and handwritten code (FastFlow + GSPARLIB) of the Mandelbrot Streaming application for the three workloads presented in Section 5.6.1, using CUDA (left column) and OpenCL (right column) backends. The lines in the graphs of Figure 5.15 represent the execution time (the lower, the better) of the handwritten code (as a blue dashed line with square markers) and automatic code generation (as an orange dot-dashed line with round markers), both related to the left Y axis with different number of parallel workers (X axis). As we mentioned in Section 5.6, the number of parallel workers in the X axis does not necessarily represent the number of active threads in the system. Rather than this, it represents the number of parallel workers in the replicated stage, and there are usually other threads dedicated for the sequential stream stages. Black error-bars are drawn in each marker. The gray bars in Figure 5.15 are related to the right Y axis and present the percentage difference in execution time of the SPar version compared to the handwritten code.

The performance of the SPar version is the same data presented in Section 5.6.1. Overall, the SPar (automatic generated) version presented lower execution times than the FastFlow+GSPARLIB (handwritten code). In fact, for the first row of graphs, representing the small workload, the biggest performance difference is 16.7% lower execution time for SPar version with six parallel workers for the CUDA backend (Figure 5.15a) and 18.2% lower execution time for the SPar version with four parallel workers for the OpenCL backend (Figure 5.15b). For the second row of graphs, which represents the medium workload, SPar also presented lower execution times in the biggest difference: 8.5% lower execution time with five parallel workers for the CUDA backend (Figure 5.15c) and 18.2% lower execution time with two parallel workers for the OpenCL backend (Figure 5.15d). Finally, in the large workload SPar presented 10.8% lower execution time with five parallel workers for the CUDA backend (Figure 5.15e), as well as 21.2% lower execution time with two parallel workers for the OpenCL backend (Figure 5.15f).

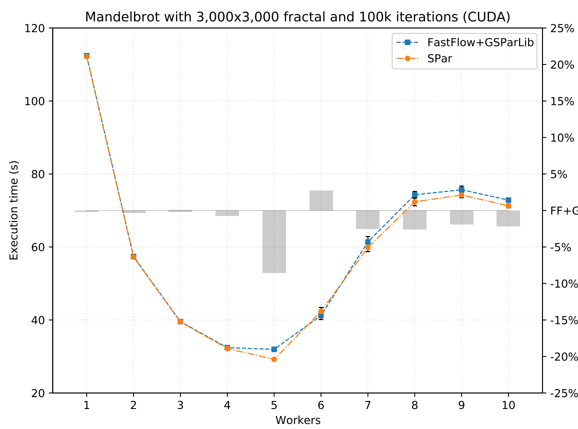
In the handwritten implementations, we followed the good programming practices that advise the reusing of allocated objects. These objects represent the GSPARLIB `Map` object parameters that are not changing for each stream item (such as the fractal size and number of iterations). The parameters are set only once before cloning an instance of the `Map` pattern object and are shared among all these cloned `Map` objects. Currently, we do not implemented so that the SPar knows what `Map` object parameters are changed between the different stream items. Thus all `Map` object parameters are set exclusively for each stream



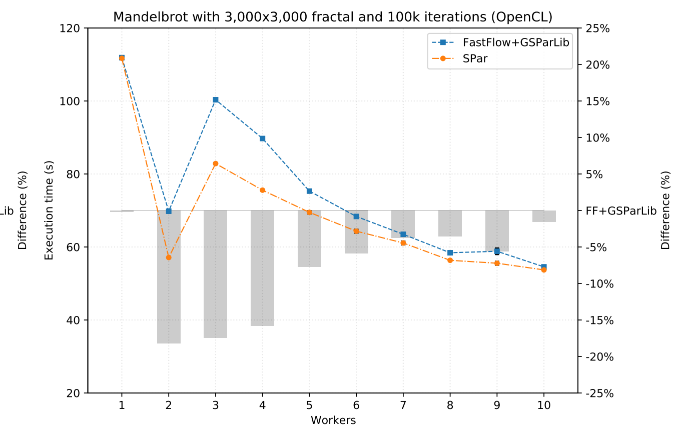
(a) Small workload with CUDA backend.



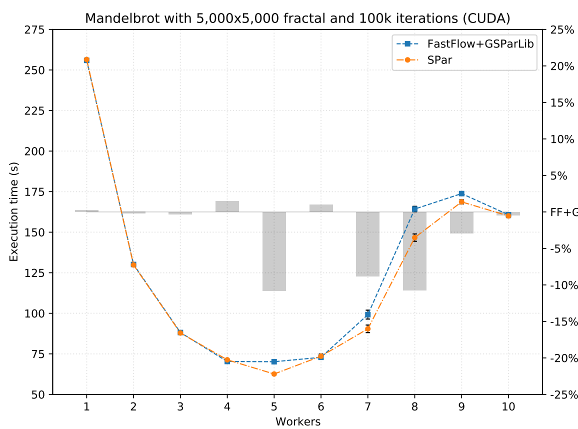
(b) Small workload with OpenCL backend.



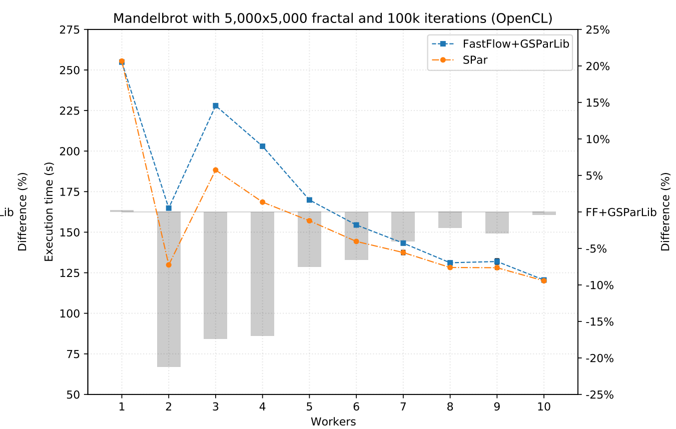
(c) Medium workload with CUDA backend.



(d) Medium workload with OpenCL backend.



(e) Large workload with CUDA backend.



(f) Large workload with OpenCL backend.

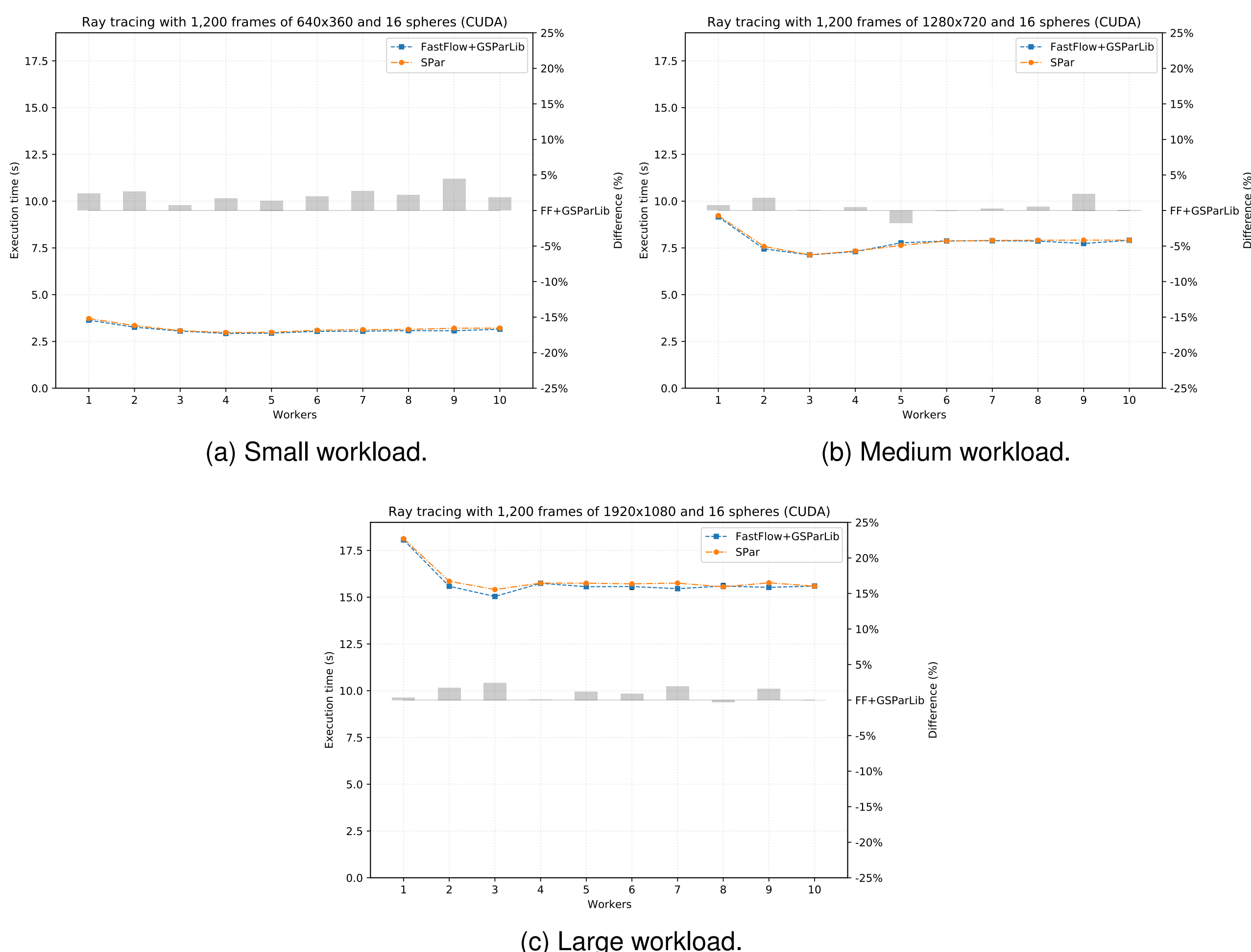
Figure 5.15: Performance difference of code generation in Mandelbrot Streaming.

item and released in the replicated stage after running the pattern. Reusing the allocated `Map` parameter objects requires less programming effort, however, it is harder to implement in the compiler for generating the code. Therefore, our generated codes are not reusing the `Map` parameter objects. Unexpectedly, we found out that the best performance is achieved by not reusing the parameters, which explains why the SPar version presents better performance than the handwritten code. This is easily perceived in the OpenCL versions, on which

the NVIDIA OpenCL driver only releases the GPU resources after all OpenCL objects are released (as discussed in Section 4.8).

5.7.2 Ray Tracing

Figure 5.16 presents a performance comparison between automatic code generation (SPar) and handwritten code (FastFlow + GSPARLIB) of the ray tracing application for the three workloads presented in Section 5.6.2 using the GSPARLIB's CUDA backend. We did not implement this application in the OpenCL backend, as discussed in Section 5.6.2. The performance of the SPar version is the same data presented in Section 5.6.1.



(a) Small workload.

(b) Medium workload.

(c) Large workload.

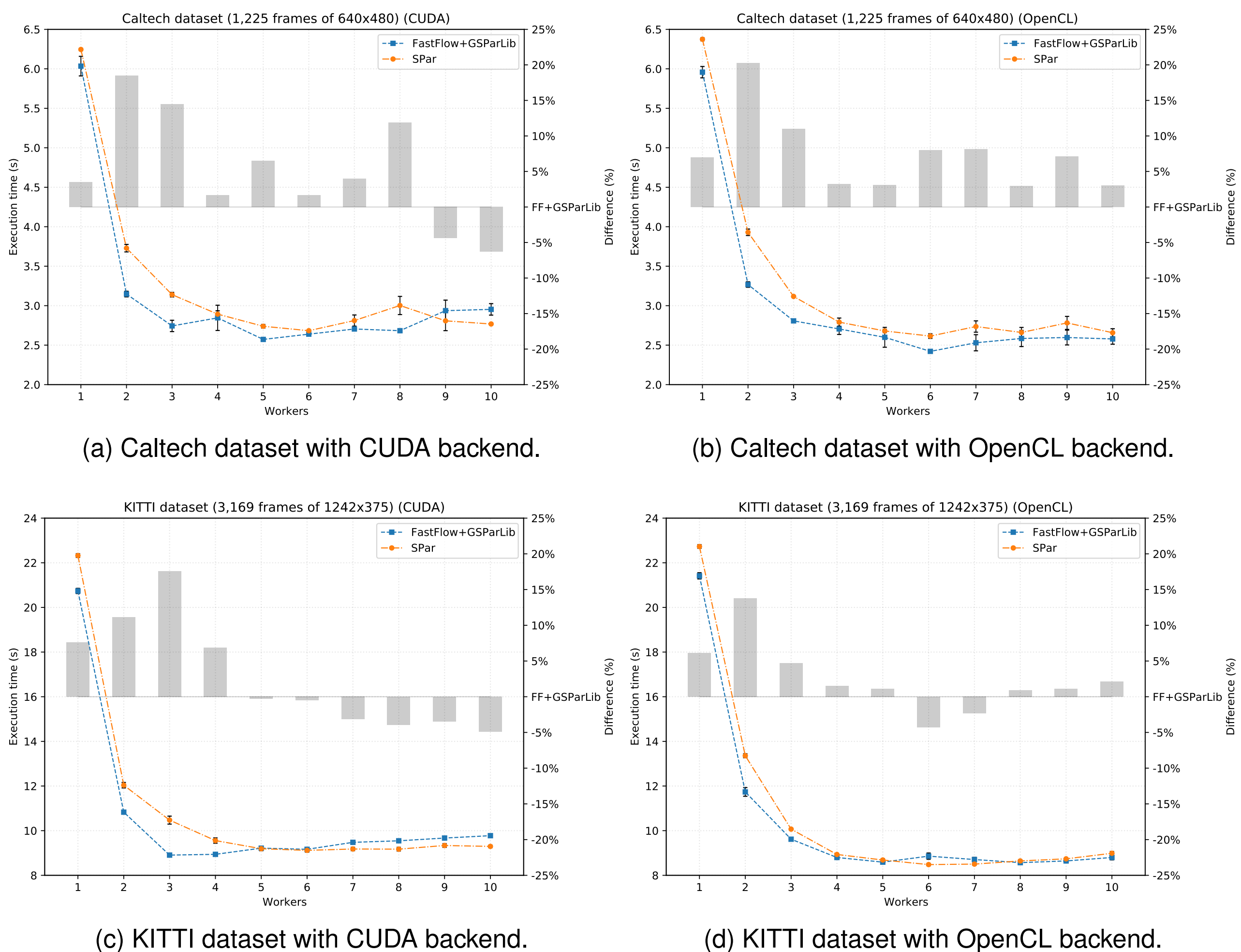
Figure 5.16: Performance difference of code generation of the ray tracing application with the CUDA backend.

In this application, the automatic code generation (SPar) presented higher execution times than the handwritten code for all workloads, with the only exception of the medium workload with five parallel workers. The biggest differences for the small (Figure 5.16a),

medium (Figure 5.16b), and large (Figure 5.16c) workloads are respectively 4.5%, 2.4%, and 2.5% higher execution times of the SPar versions with respect to the handwritten code.

5.7.3 Lane Detection

Figure 5.17 presents a performance comparison between automatic code generation (SPar) and handwritten code (FastFlow + GSPARLIB) of the lane detection application for the two datasets (Caltech and KITTI) presented in Section 5.6.3, using the GSPARLIB's CUDA and OpenCL backends. The performance of the SPar version is the same data presented in Section 5.6.3.



(a) Caltech dataset with CUDA backend.

(b) Caltech dataset with OpenCL backend.

(c) KITTI dataset with CUDA backend.

(d) KITTI dataset with OpenCL backend.

Figure 5.17: Performance difference of code generation in lane detection.

For the Caltech dataset (first row of graphs in Figure 5.17) the biggest differences between the automatic code generation and handwritten code are using two parallel workers: 18.5% higher execution time for the CUDA backend and 20.3% higher execution time for the OpenCL backend. For the KITTI dataset (second row of graphs in Figure 5.17), the biggest differences between the automatic code generation and handwritten code are 17.6% higher

execution time for the CUDA backend with three parallel workers and 13.8% higher execution time for the OpenCL backend with two parallel workers.

We noticed some performance differences between the automatic code generation and handwritten code for the lane detection application, mainly using a lower number of workers. However, it requires more investigations to understand these differences.

5.8 SPar Programmability Considerations

In this section we briefly discuss the programmability benefits of using SPar. Figure 5.18 shows how the vector sums sample application discussed in Section 4.2 can be annotated using the SPar language. Using our novel attributes and transformation rules, the SPar compiler is now able to automatically perform the transformations shown in Figure 4.3 (using the stream parallelism transformation rules) as well as the transformations shown in Figure 4.4 (using the data parallelism transformation rules). The programmer adding only three annotations in the sequential code can in the most cases take advantage of stream (multi-core CPU) and data parallelism (many-core GPU) on stream processing applications.

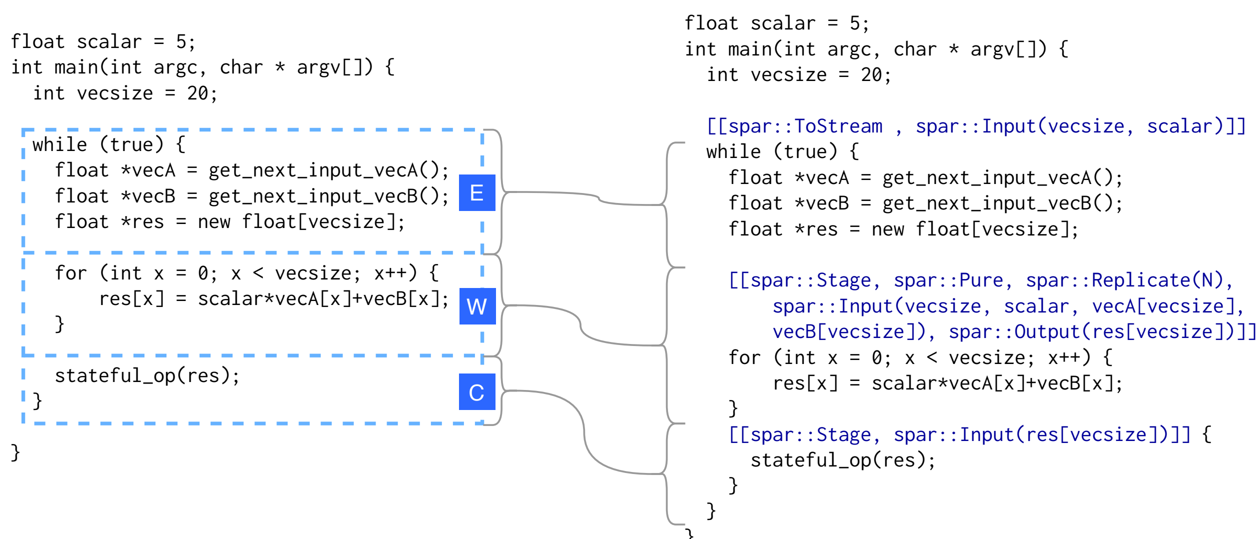


Figure 5.18: Applying SPar in sequential code.

We briefly analyze the physical Source Lines of Code (SLOC) added to the sequential version of some applications as a rough measure of the programmer's productivity using SPar compared to handwritten code of the same applications using FastFlow and GSParLib. We already discussed the programmability aspects of GSParLib in Section 4.6. Thus, the SLOC would be bigger if we were to use another GPU programming API instead of GSParLib together with FastFlow. Table 5.10 present a summary of the physical SLOC of some applications. The metric was extracted using David A. Wheeler's 'SLOCCount' [Whe16].

Table 5.10: Physical SLOC comparison.

Application	*Sequential*	*FastFlow+*GSPARLIB	*SPar*
Vectors sums[a]	13	59 (+354%)	17 (+31%)
Mandelbrot Streaming	48	131 (+173%)	56 (+17%)
Ray tracing	455	626 (+38%)	463 (+2%)
Lane detection	300	452 (+51%)	312 (+4%)

[a] This is the sample application shown in Figures 4.3, 4.4, and 5.18.

The sequential and SPar versions of the Vectors sums application is shown in the Figure 5.18. For this application, the SPar version adds only four extra lines: one `ToStream` annotation, two `Stage` annotations, and one closing curly bracket to delimit the code region of the last `Stage` (which represents the Farm's Collector). The FastFlow+GSPARLIB version is shown in the right side of Figure 4.4, which presents 3.5× more SLOC than the sequential version.

The Mandelbrot Streaming application was discussed in Section 5.6.1. The streaming region of the SPar version of this application is presented in Listing 5.1. It is only eight lines longer than the sequential version, which are for adding the SPar annotations and managing the `workers` variable. The FastFlow+GSPARLIB version includes the definition of the three Farm stages (Emitter, Worker, and Collector) and the definition of GSPARLIB `Map` pattern. It requires significant code refactoring to allow the exploitation of parallelism in multi-core CPU and many-core GPU architectures.

The ray tracing application was discussed in Section 5.6.2. The streaming region of the SPar version of this application is presented in Listing 5.3. With respect to the sequential version, it only adds four annotations and a few lines of code to manage the number of parallel workers. The FastFlow+GSPARLIB version requires 38% more SLOC with respect to the sequential version, however, most of the code had to be refactored since it is called inside the GPU kernel (referenced by the `trace` function). Therefore, it must be passed to the GSPARLIB pattern object using the `addExtraKernelCode` method.

The lane detection application was discussed in Section 5.6.3. The streaming region of the SPar version for this application is presented in Listing 5.4. Only 12 lines of code were added to exploit multi-core CPU and many-core GPU parallelism with SPar, using annotations to delimit the stream region, the two stages, and the three pure loops of computation. The FastFlow+GSPARLIB version of this application requires 51% more SLOC to exploit this heterogeneous parallelism, since it is necessary to create and call three separate instances of the GSPARLIB Map pattern.

5.9 Final Remarks

In this chapter we presented our extension of the SPar language to express data parallelism in stream processing applications. We extended the language expressiveness with novel high-level attributes and presented novel transformation rules targeting the combination of stream and data parallel patterns. These rules were implemented in the SPar compiler to automatic generate parallel code with FastFlow and GSParLib runtimes. In our tests, were able to achieve good performance improvements compared to the SPar multi-core backend, concerning throughput and latency. We also demonstrated that the automatic code generation did not incur in performance penalty with respect to the handwritten implementation using the same underlying libraries in three applications, namely Mandelbrot Streaming, ray tracing, and lane detection. The experiments were executed in a single machine equipped with a multi-core GPU and many-core GPU. It is a common computer architecture for dedicated servers and workstations.

We were able to successfully exploit stream and data parallelism by simply adding a few lines of code annotations delimiting the stream and pure code regions. In this sense, SPar is unique in the literature by allowing efficient parallelism exploitation without requiring the programmer to know architecture details. Our tests show that the programmer is able to efficiently exploit heterogeneous parallelism composed of multi-core CPU and many-core GPU using the high-level SPar abstractions. For the GPU support, SPar leverages GSParLib presented in Chapter 4, which offers a unified and pattern-based interface as well as a drive-agnostic runtime capable of using both CUDA and OpenCL.

Moreover, Figure 5.19[5] illustrates the current SPar runtimes and supported patterns. SPar generates the Pipeline and Farm stream parallel patterns using FastFlow and TBB in multi-core environments and DSParLib [Pie20] in cluster architectures. Our work adds support to Map and Reduce data parallel patterns to the SPar ecosystem using the GSParLib runtime. GSParLib focuses in heterogeneous environments composed of a multi-core CPU and one or more GPUs as co-processors. It is worth noting that we implemented data parallelism support on top of the current stream parallelism implementation.

In addition to the future works highlighted in Figure 5.19, that is implement support for other parallel patterns, the results obtained in this work offer many other research opportunities. For example, future works may improve the `Batch` attribute by batching the stream items before sending them to the batched stage for improving latency times. Another future work is the support for batching stream items in the multi-core (FastFlow/TBB) and cluster (DSParLib) runtimes to support the use of the `Batch` attribute with and without the `Pure` attribute. SPar compiler does not generate code for multi-GPU in a single computer, although the GSParLib runtime supports it. Therefore, it can be implemented a scheduler

[5] Icons made by phatplus from www.flaticon.com

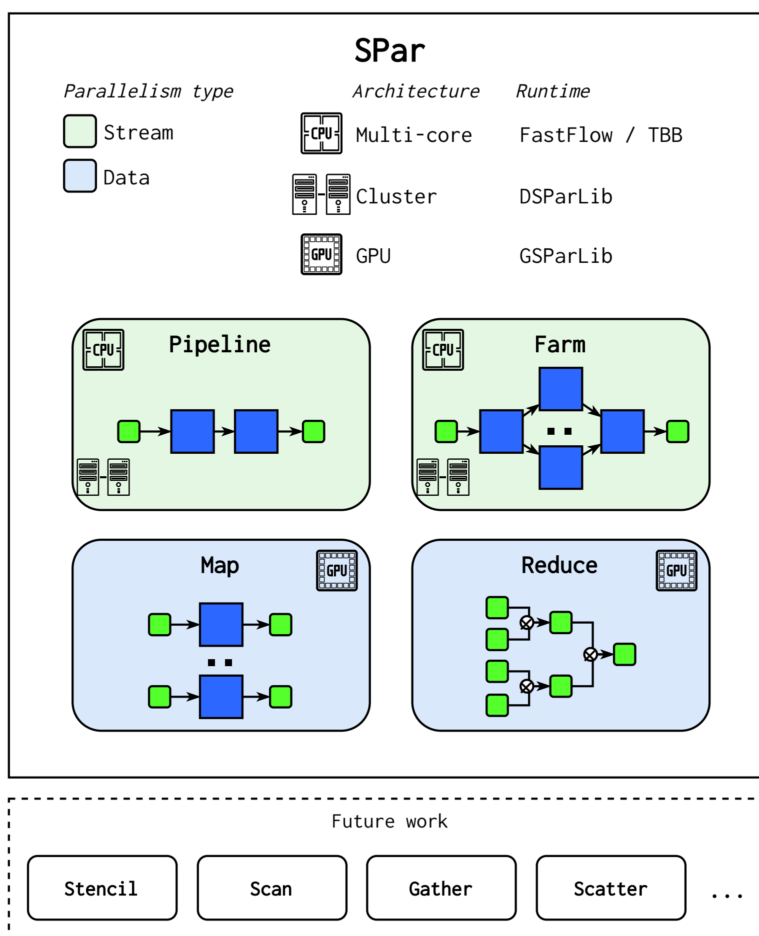


Figure 5.19: Overview of SPar runtimes and supported parallel patterns.

capable of automatically distributing the stream items to multiple GPUs. We also left as future work the support of GPU parallelism in cluster architectures, i.e. generating code using DSParLib [Pie20] and GSParLib runtimes together.

In the current SPar version, the releasing of GPU resources is performed by the worker thread right after finishing the kernel call and the memory copies. Nonetheless, in one of our tests in the Mandelbrot Streaming application, we changed the code generation to release the GPU resources in the last `Stage`, which was the Farm's Collector, and found out that there was a significant (∼50%) performance penalty in the throughput when using a low number of workers (up to seven), but also an increase in the throughput when using more than eight workers. This suggests an interesting alternative to better balance the workload between workers and collector threads and may be explored by future works.

6. CONCLUSION

In this work we addressed the problem of exploiting parallelism on heterogeneous computer architectures composed of multi-core CPUs and many-core GPUs when programming stream processing applications. The rise of heterogeneous computer architectures posed challenges to application developers. The current *de facto* standard APIs (CUDA and OpenCL) are still too low-level, requiring to learn hardware architecture details in order to efficiently exploit GPU parallelism. The lack of high-level abstractions for these computer architectures regarding stream processing applications highlight the importance of this work, specially given that both architecture and application are increasingly pervasive in the current technology landscape. As the domain-specific language extensions proposed and implemented in SPar provided high-level abstractions for exploiting multi-core CPU and many-core GPU parallelism in the stream processing applications, including the developing of GSPARLIB used as the runtime library for the generated code, we were able to answer the question that drove this research work.

Based on the literature, we conclude that GSPARLIB is a novel structured parallel programming API for GPU programming, distinguished by its unified API and driver-agnostic runtime that allows programmers to transparently switch between the *de facto* standards CUDA and OpenCL drivers. GSPARLIB provides its layered API without requiring a custom compiler, and offers essential features for GPU parallelism when parallel programming stream processing applications like thread-safety and batching. We initially provide Map and Reduce parallel patterns in its high-level API. However, it also allows the programmer customize computational patterns by using the lower-level API, which serves as a thin and unified C++ object-oriented abstraction layer over CUDA and OpenCL drivers. Our experiments shows that the abstractions provided by GSPARLIB does not present high performance penalties when compared to related APIs from industry and academia, including state-of-the-art tools and lower-level APIs.

Moreover, important and novel scientific contributions for high-level parallel programming abstractions were possible due to the: 1) extension of the SPar language expressiveness, adding three novel C++11 attributes, namely `Pure`, `Reduce`, and `Batch`; 2) creation of new definitions and parallel pattern-based transformation rules, combining stream and data parallel patterns; 3) implementation of these transformation rules in the SPar source-to-source compiler, generating parallel code that combines FastFlow and GSPARLIB runtimes; and 4) performance and programmability evaluation with real-world stream processing applications, highlighting that the abstractions are lightweight compared to handwritten codes while it leverages simple programming, few code refactoring, and high-level application domain concepts. To the best of our knowledge, thanks to these efforts, SPar is now unique in the literature by entirely abstracting architecture details for exploiting parallelism on multi-core,

cluster, and heterogeneous computer architectures when programming stream processing applications.

The results obtained in this work may be improved in different ways. We describe the main research opportunities in the following items:

- **Other heterogeneous computer architectures in GSPARLIB and SPar.** GSPARLIB's Driver API currently supports only GPU architectures using CUDA and OpenCL drivers. Nonetheless, this API may be extended to support different accelerator architectures, such as FPGA and Intel Many Integrated Core (MIC) architectures. Since the OpenCL code for GPU cannot be used to exploit FPGA architectures, it involves implementing OpenCL for FPGA. The SPar language and compiler may also be extended to provide support for different heterogeneous computer architectures.
- **Optimizations in GSPARLIB.** We focused in developing GSPARLIB's layered unified API and driver-agnostic runtime, thus little effort was dedicated into developing code optimizations. Future works may implement optimizations regarding data locality considering the GPU memory architecture, support for unified memory, and reduce branch divergence.
- **Other parallel patterns in GSPARLIB.** The GSPARLIB's Pattern API may be extended to support other parallel patterns focused in data parallelism, such as Stencil, Scan, Gather, and Scatter. When implementing these patterns, it is also important support streaming-related features such as thread-safety and batching.
- **Definitions and transformation rules for other parallel patterns.** Currently SPar generates the Pipeline and Farm parallel patterns for stream parallelism as well as Map and Reduce parallel patterns for data parallelism. With the creation of new definitions and transformation rules, it may be possible to generate other parallel patterns without changing SPar's syntax. However, it may be necessary to extend the SPar expressiveness if the current attributes does not convey enough information for the code generation.
- **Service Level Objectives (SLO) in heterogeneous computer architectures.** Our work provide tools for extending the work of autonomic computing and high-level SLO in SPar [VGS+18, GSV+18, GVS+19]. Future works may implement adaptive strategies using the batch size [SRG+20] or configuring the size of the GPU kernel thread grid.
- **Hybrid parallelism support in SPar.** In its current form, the SPar compiler always offload pure code regions to the GPU accelerator. Future works may explore algorithms to automatically chose the best parallel device for each code region by analyzing code characteristics and querying the device properties. The first step towards this goal of hybrid parallelism is to implement the support for multi-GPU using an algorithm for automatic scheduling of GPU kernels among the available devices.

6.1 List of published papers

- **Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units**. *Concurrency and Computation: Practice and Experience (CCPE)*, 2020 [SRG+20].
In this paper we tackle the problem of adaptive micro-batching for GPU parallelism in stream processing applications with focus in high-level latency objectives [VGS+18, GSV+18]. However, the code generation of adaptive micro-batching size for the `Batch` attribute is left as future work due to its complexity.
- **Stream Processing on Multi-cores with GPUs: Parallel Programming Models' Challenges**. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Workshop on Parallel Programming Models (MPP)*, 2019 [RSG+19].
In this work we showed the importance of using multiple CPU threads for each GPU device and present an overview of common programming challenges faced by heterogeneous parallel programmers to combine multi-core CPU and many-core GPU parallelism. Many lessons-learn were take into account for the design choices in this master thesis.
- **High-Level Stream Parallelism Abstractions with SPar Targeting GPUs**. *International Conference on Parallel Computing (ParCo)*, 2019 [RGDF19].
This paper continues our previous work of extending the SPar language. Here we define the requirements of the `Pure` attribute and present a subset of the novel definitions and pattern-based transformation rules targeting the Map parallel pattern.
- **Proposta de Suporte ao Paralelismo de GPU na SPar**. *19th Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)*, 2019 [RGF19].
This work present our ongoing work during the design phase of the novel `Pure` and `Batch` attributes for the SPar language. It is focused in the language expressiveness and does not present any implementation or performance tests.
- **Paralelização do Dedup para Sistemas Multi-core com GPUs**. *19th Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)*, 2019 [SRG19].
In this work we implemented the Dedup application with the LZSS compression algorithm to exploit an heterogeneous computer architecture comprised of multi-core CPUs and many-core GPUs using SPar, CUDA, and OpenCL. We were not able to properly exploit the GPU parallelism because the LZSS segmentation algorithm generates very small blocks, which provides us indications for the need of the `Batch` attribute.
- **Mandelbrot Streaming para Sistemas Multi-core com GPUs**. *19th Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)*, 2019 [SSB+19].

In this work we implemented the Mandelbrot Streaming application using SPar, FastFlow, and TBB for multi-core parallelism, and CUDA for GPU parallelism. This and the [SRG19] work were important for us to understand the challenges of combining multi-core and GPU parallelism.

REFERENCES

[ABI18] ABI Research. “ABI Research Forecasts 8 Million Vehicles to Ship with SAE Level 3, 4 and 5 Autonomous Technology in 2025”. Source: https://www.abiresearch.com/search/press/5231/, 12 nov 2020.

[ADD+18] Aldinucci, M.; Danelutto, M.; Drocco, M.; Kilpatrick, P.; Misale, C.; Peretti Pezzi, G.; Torquati, M. “A parallel pattern for iterative stencil + reduce”, *The Journal of Supercomputing*, vol. 74–11, Nov. 2018, pp. 5690–5705.

[ADKT17] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “FastFlow: high-level and efficient streaming on multi-core”. John Wiley & Sons, 2017, 1 ed., chap. 13, pp. 261–280.

[ADM+09] Aldinucci, M.; Danelutto, M.; Meneghin, M.; Torquati, M.; Kilpatrick, P. “Efficient streaming applications on multi-core with FastFlow: The biosequence alignment test-bed”. In: Proceedings of the International Conference on Parallel Computing, 2009, pp. 273–280.

[AGDF20] Araujo, G. A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Efficient NAS Parallel Benchmark Kernels with CUDA”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2020, pp. 9–16.

[AGT14] Andrade, H.; Gedik, B.; Turaga, D. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 1 ed., 558p.

[Aly08] Aly, M. “Real time detection of lane markers in urban streets”. In: Proceedings of the IEEE Symposium on Intelligent Vehicle, 2008, pp. 7–12.

[Ama17] Amazon Lumberyard. “Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)”. Source: https://developer.nvidia.com/orca/amazon-lumberyard-bistro, 12 nov 2020.

[AMD14] AMD. “Bolt 1.3: C++ template library with support for OpenCL”, 2014, Version 1.3, Source: https://hsa-libraries.github.io/Bolt/html/.

[APD+15] Aldinucci, M.; Pezzi, G. P.; Drocco, M.; Spampinato, C.; Torquati, M. “Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern”, *The International Journal of High Performance Computing Applications*, vol. 29–4, Feb. 2015, pp. 461–472.

[App68] Appel, A. “Some techniques for shading machine renderings of solids”. In: Proceedings of the AFIPS Spring Joint Computer Conference, 1968, pp. 37–45.

[ATAF14] Augonnet, C.; Thibault, S.; Aumage, O.; Furmento, N. “StarPU: seamless computations among CPUs and GPUs”. Source: https://www.x.org/wiki/Events/XDC2014/XDC2014ThibaultStarPU, 12 nov 2020.

[ATNW11] Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”, *Concurrency and Computation: Practice and Experience*, vol. 23–2, Feb. 2011, pp. 187–198.

[Aug11] Augonnet, C. “Scheduling Tasks over Multicore machines enhanced with acelerators: a Runtime System’s Perspective”, Ph.D. Thesis, Efficient runtime systems for parallel architectures - Inria Bordeaux - Université de Bordeaux, Bordeaux, France, 2011, 228p.

[Bal81] Ballard, D. “Generalizing the Hough transform to detect arbitrary shapes”, *Pattern Recognition*, vol. 13–2, Sep. 1981, pp. 111–122.

[Bal08] Balagurusamy, E. “Object Oriented Programming with C++”. Tata McGraw-Hill, 2008, 4 ed., 637p.

[Bav19] Bavoil, L. “The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload”. Source: https://developer.nvidia.com/blog/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/, 12 nov 2020.

[BB12] Barde, M. P.; Barde, P. J. “What to use to express the variability of data: Standard deviation or standard error of mean?”, *Perspectives in Clinical Research*, vol. 3–3, Jul. 2012, pp. 113–116.

[Bel20] Belton, P. “Get ready for the ’holy grail’ of computer graphics”. Source: https://www.bbc.com/news/business-52541218, 12 nov 2020.

[Ben18] Bendersky, E. “Affine transformations”. Source: https://eli.thegreenplace.net/2018/affine-transformations/, 12 nov 2020.

[BH10] Brodtkorb, A. R.; Hagen, T. R. “A Comparison of Three Commodity-Level Parallel Architectures: Multi-core CPU, Cell BE and GPU”. In: *Mathematical Methods for Curves and Surfaces*, Springer, 2010, vol. 5862, pp. 70–80.

[BJB+20] Brown, C.; Janjic, V.; Barwell, A. D.; Garcia, J. D.; MacKenzie, K. “Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++”, *International Journal of Parallel Programming*, vol. 48–4, Aug. 2020, pp. 603–625.

[BRS10] Baskaran, M. M.; Ramanujam, J.; Sadayappan, P. “Automatic C-to-CUDA Code Generation for Affine Programs”. In: *Compiler Construction*, Springer, 2010, vol. 6011, pp. 244–263.

[Can86] Canny, J. “A Computational Approach to Edge Detection”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8–6, Nov. 1986, pp. 679–698.

[CGM14] Cheng, J.; Grossman, M.; McKercher, T. “Professional CUDA C Programming”. John Wiley & Sons, 2014, 1 ed., 528p.

[CJ09] Chakravarthy, S.; Jiang, Q. “Stream Data Processing: A Quality of Service Perspective. Modeling, Scheduling, Load Shedding, and Complex Event Processing”. Springer, 2009, vol. 36, 324p.

[CKM19] Cho, H. D.; Kwon, O.; Midkiff, S. P. “HDArray: Parallel Array Interface for Distributed Heterogeneous Devices”. In: *Languages and Compilers for Parallel Computing*, Springer, 2019, vol. 11882, pp. 176–184.

[Col89] Cole, M. I. “Algorithmic Skeletons: Structured Management of Parallel Computation”. Cambridge: Pitman/MIT Press, 1989, 1 ed., 132p.

[Col04] Cole, M. “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming”, *Parallel Computing*, vol. 30–3, Mar. 2004, pp. 389–406.

[CTS14] Carter Edwards, H.; Trott, C. R.; Sunderland, D. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”, *Journal of Parallel and Distributed Computing*, vol. 74–12, Dec. 2014, pp. 3202–3216.

[DARG13] Demidov, D.; Ahnert, K.; Rupp, K.; Gottschling, P. “Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries”, *SIAM Journal on Scientific Computing*, vol. 35–5, Sep. 2013, pp. C453–C472.

[DDMT18] Danelutto, M.; De Matteis, T.; Mencagli, G.; Torquati, M. “Data stream processing via code annotations”, *The Journal of Supercomputing*, vol. 74, Nov. 2018, pp. 5659–5673.

[DGS+16] Danelutto, M.; Garcia, J. D.; Sanchez, L. M.; Sotomayor, R.; Torquati, M. “Introducing Parallelism by Using REPARA C++11 Attributes”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2016, pp. 354–358.

[DM98] Dagum, L.; Menon, R. “OpenMP: an industry standard API for shared-memory programming”, *IEEE Computational Science and Engineering*, vol. 5–1, Jan. 1998, pp. 46–55.

[DM16] De Matteis, T.; Mencagli, G. “Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing”, *ACM SIGPLAN Notices*, vol. 51–8, Feb. 2016, pp. 1–12.

[DM17] De Matteis, T.; Mencagli, G. “Proactive elasticity and energy awareness in data stream processing”, *Journal of Systems and Software*, vol. 127, May. 2017, pp. 302–319.

[EAB+20] Ejjaaouani, K.; Aumage, O.; Bigot, J.; Méhrenberger, M.; Murai, H.; Nakao, M.; Sato, M. “InKS: a Programming Model to Decouple Algorithm from Optimization in HPC Codes”, *The Journal of Supercomputing*, vol. 76, Jun. 2020, pp. 4666–4681.

[EK19] Ernstsson, A.; Kessler, C. “SkePU: Introduction & Tutorial”. Source: https://skepu.github.io/docs/MCC2019-SkePU-Tutorial.pdf, 12 nov 2020.

[EK20] Ernstsson, A.; Kessler, C. “Multi-Variant User Functions for Platform-Aware Skeleton Programming”. In: Proceedings of the International Conference on Parallel Computing, 2020, pp. 475–484.

[ELK18] Ernstsson, A.; Li, L.; Kessler, C. “SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems”, *International Journal of Parallel Programming*, vol. 46–1, Feb. 2018, pp. 62–80.

[EMBS17] Earl, C.; Might, M.; Bagusetty, A.; Sutherland, J. C. “Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations”, *Journal of Systems and Software*, vol. 125, Mar. 2017, pp. 389–400.

[End17] Endsley, M. R. “Autonomous Driving Systems: A Preliminary Naturalistic Study of the Tesla Model S”, *Journal of Cognitive Engineering and Decision Making*, vol. 11–3, Feb. 2017, pp. 225–238.

[FHTW20] Fang, J.; Huang, C.; Tang, T.; Wang, Z. “Parallel programming models for heterogeneous many-cores: a comprehensive survey”, *CCF Transactions on High Performance Computing*, vol. 2, Jan. 2020, pp. 382–400.

[Fow10] Fowler, M. “Domain Specific Languages”. Addison-Wesley Professional, 2010, 1 ed., 640p.

[FSH04] Fatahalian, K.; Sugerman, J.; Hanrahan, P. M. “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication”. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2004, pp. 133–137.

[Gar19] Gartner. “Gartner Forecasts More Than 740,000 Autonomous-Ready Vehicles to Be Added to Global Market in 2023”. Source: https://www.gartner.com/en/newsroom/press-releases/2019-11-14-gartner-forecasts-more-than-740000-autonomous-ready-vehicles-to-be-added-to-global-market-in-2023, 12 nov 2020.

[GCC20] GCC. “Offloading Support in GCC - GCC Wiki”. Source: https://gcc.gnu.org/wiki/Offloading, 12 nov 2020.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, Mar. 2017, pp. 1740005.

[GF11] Griebler, D.; Fernandes, L. G. “Padrões e Frameworks de Programação Paralela em Arquiteturas Multi-Core”, Technical Report, Faculty of Informatics - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2011, 55p.

[GF17] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: Proceedings of the International Conference on Parallel Computing, 2017, pp. 563–572.

[GHDF17] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-Level Parallelism Abstractions for Video Applications with SPar”. In: Proceedings of the International Conference on Parallel Computing, 2017, pp. 698–707.

[GHDF18] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, vol. 47–1, Feb. 2018, pp. 253–271.

[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software ”. Addison-Wesley, 1994, 1 ed., 395p.

[GHL+17] Griebler, D.; Hoffmann, R. B.; Loff, J.; Danelutto, M.; Fernandes, L. G. “High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications”. In: Proceedings of the XVIII Symposium on High Performance Computing Systems, 2017, pp. 16–27.

[Gla89] Glassner, A. S. “An Introduction to Ray Tracing”. Academic Press, 1989, 1 ed., 368p.

[GLSU13] Geiger, A.; Lenz, P.; Stiller, C.; Urtasun, R. “Vision meets Robotics: The KITTI Dataset”, *International Journal of Robotics Research*, vol. 32–11, Sep. 2013, pp. 1231–1237.

[GM12] Gregory, K.; Miller, A. “C++ AMP”. Sebastopol, United States of America: O’Reilly Media, 2012, 1 ed., 356p.

[GP92] Girkar, M.; Polychronopoulos, C. D. “Automatic Extraction of Functional Parallelism from Ordinary Programs”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 3–2, Mar. 1992, pp. 166–178.

[Gri16] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculty of Informatics - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2016, 243p.

[GSV+18] Griebler, D.; Sensi, D. D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service Level Objectives via C++11 Attributes”. In: *Euro-Par 2018: Parallel Processing Workshops*, Springer, 2018, vol. 11339, pp. 745–756.

[GTS11] Guo, J.; Thiyagalingam, J.; Scholz, S.-B. “Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler”. In: Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming, 2011, pp. 15–24.

[GVS+19] Griebler, D.; Vogel, A.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Simplifying and implementing service level objectives for stream parallelism”, *Journal of Supercomputing*, vol. 76, Jun. 2019, pp. 4603–4628.

[HA09] Han, T. D.; Abdelrahman, T. S. “*hi*CUDA: a high-level directive-based language for GPU programming”. In: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 52–61.

[HA11] Han, T. D.; Abdelrahman, T. S. “*hi*CUDA: High-Level GPGPU Programming”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 22–1, Jan. 2011, pp. 78–90.

[HAM19] Haines, E.; Akenine-Möller, T. “Ray Tracing Gems”. Apress, 2019, 1 ed., 607p.

[Har07] Harris, M. “Optimizing Parallel Reduction in CUDA”. Source: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, 12 nov 2020.

[Har12a] Harris, M. “How to Optimize Data Transfers in CUDA C/C++”. Source: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, 12 nov 2020.

[Har12b] Harris, M. “How to Overlap Data Transfers in CUDA C/C++”. Source: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/, 12 nov 2020.

[Har15] Harris, M. “A Brief History of General-Purpose Computation on GPUs”. Source: https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf, 12 nov 2020.

[HCSL02] Harris, M. J.; Coombe, G.; Scheuermann, T.; Lastra, A. “Physically-based visual simulation on graphics hardware”. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2002, pp. 109–118.

[HG14] Haidl, M.; Gorlatch, S. "PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14". In: Proceedings of the LLVM Compiler Infrastructure in HPC, 2014, pp. 1–11.

[HGDF20] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. "Stream Parallelism Annotations for Multi-Core Frameworks". In: Proceedings of the XXIV Brazilian Symposium on Programming Languages, 2020, pp. 48–55.

[HLLR14] Hillel, A. B.; Lerner, R.; Levi, D.; Raz, G. "Recent progress in road and lane detection: a survey", *Machine Vision and Applications*, vol. 25–3, Apr. 2014, pp. 727–745.

[Hof20] Hoffmann, R. B. "Stream Parallelism Annotations for Autonomic OpenMP Code Generation", Technical Report, School of Technology - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2020, 55p.

[HSA18a] HSA Foundation. "HSA Platform System Architecture Specification", 2018, Version 1.2.

[HSA18b] HSA Foundation. "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG)", 2018, Version 1.2.

[HSA18c] HSA Foundation. "HSA Runtime Programmer's Reference Manual", 2018, Version 1.2.

[HSS+14] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. O. "A catalog of stream processing optimizations", *ACM Computing Surveys*, vol. 46–4, Mar. 2014, pp. 46:1–46:34.

[HSW+11] Hormati, A. H.; Samadi, M.; Woh, M.; Mudge, T.; Mahlke, S. "Sponge: Portable Stream Programming on Graphics Engines". In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 381–392.

[Int17] International Organization for Standardization. "ISO/IEC 14882:2017 - Programming languages – C++", 2017, Source: https://www.iso.org/standard/68564.html.

[KDW10] Kestur, S.; Davis, J. D.; Williams, O. "BLAS Comparison on FPGA, CPU and GPU". In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI, 2010, pp. 288–293.

[Khr18] The Khronos Group. "The OpenCL™ Specification", 2018, Version 2.2-7.

[Khr20a] Khronos Group. “Conformant Products - OpenCL”. Source: https://www.khronos.org/conformance/adopters/conformant-products/opencl, 12 nov 2020.

[Khr20b] The Khronos Group. “The OpenCL™ Specification”, 2020, Version v3.0.5.

[Khr20c] Khronos® SYCL™ Working Group. “SYCL™ Specification”, 2020, version 1.2.1, Source: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

[KMSZ15] Kaeli, D.; Mistry, P.; Schaa, D.; Zhang, D. P. “Heterogeneous Computing with OpenCL 2.0”. Morgan Kaufmann, 2015, 3 ed., 307p.

[KmWH16] Kirk, D. B.; mei W. Hwu, W. “Programming Massively Parallel Processors”. Morgan Kaufmann, 2016, 2 ed., 576p.

[KP08] Kammel, S.; Pitzer, B. “Lidar-based lane marker detection and mapping”. In: Proceedings of the IEEE Symposium on Intelligent Vehicle, 2008, pp. 1137–1142.

[Lar18] Larkin, J. “OpenMP on GPUs, First Experiences and Best Practices”. Source: https://on-demand.gputechconf.com/gtc/2018/presentation/s8344-openmp-on-gpus-first-experiences-and-best-practices.pdf, 10 nov 2020.

[LB20] Lonsdorf, B.; Benkort, M. “Professor Frisby’s Mostly Adequate Guide to Functional Programming”. Source: https://mostly-adequate.gitbooks.io/mostly-adequate-guide/, 01 nov 2020.

[LD09] Lastovetsky, A. L.; Dongarra, J. J. “High-performance Heterogeneous Computing”. John Wiley & Sons, 2009, 1 ed., 267p.

[Led16] Ledur, C. “GMaVis: A Domain-Specific Language for Large-Scale Geospatial Data Visualization Supporting Multi-core Parallelism”, Master’s Thesis, Faculty of Informatics - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2016, 143p.

[LI15] Lake, A. T.; Ioffe, R. M. “The Generic Address Space in OpenCL™ 2.0”. Source: https://software.intel.com/content/www/us/en/develop/articles/the-generic-address-space-in-opencl-20.html, 12 nov 2020.

[LM01] Larsen, E. S.; McAllister, D. “Fast matrix multiplies using graphics hardware”. In: Proceedings of the ACM/IEEE conference on Supercomputing, 2001, pp. 1–6.

[LTO+12] Lee, J.; Tran, M. T.; Odajima, T.; Boku, T.; Sato, M. “An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters”. In: *Euro-Par 2011: Parallel Processing Workshops*, Springer, 2012, vol. 7155, pp. 429–439.

[Lut15] Lutz, K. “Boost.Compute”. Source: https://github.com/boostorg/compute, 12 nov 2020.

[Lö20] Löff, J. H. “Aumentando a Expressividade e Melhorando a Geração de Código Paralelo para o Paradigma de Paralelismo de Stream em Arquiteturas Multi-core”, Technical Report, School of Technology - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2020, 76p.

[MGA+17] Mendonça, G.; Guimarães, B.; Alves, P.; Pereira, M.; Araújo, G.; Pereira, F. M. Q. “DawnCC: Automatic Annotation for Data Parallelism and Offloading”, *ACM Transactions on Architecture and Code Optimization*, vol. 14–2, May. 2017, pp. 13:1–25.

[MGM+12] Munshi, A.; Gaster, B. R.; Mattson, T. G.; Fung, J.; Ginsburg, D. “OpenCL Programming Guide”. Addison-Wesley, 2012, 1 ed., 603p.

[Mic17] Microsoft. “Introduction to PLINQ”. Source: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq, 12 nov 2020.

[Moo65] Moore, G. E. “Cramming more components onto integrated circuits”, *Electronics*, vol. 38–8, Apr. 1965, pp. 114–117.

[MRR12] McCool, M.; Robison, A. D.; Reinders, J. “Structured Parallel Programming: Patterns for Efficient Computation”. Morgan Kaufmann, 2012, 1 ed., 432p.

[MSM04] Mattson, T. G.; Sanders, B. A.; Massingill, B. L. “Patterns for Parallel Programming”. Pearson Education, 2004, 1 ed., 384p.

[MYM+12] Malcolm, J.; Yalamanchili, P.; McClanahan, C.; Venugopalakrishnan, V.; Patel, K.; Melonakos, J. “ArrayFire: a GPU acceleration platform”. In: Proceedings of SPIE Modeling and Simulation for Defense Systems and Applications VII, 2012, pp. 84030A.

[NMS+14] Nakao, M.; Murai, H.; Shimosaka, T.; Tabuchi, A.; Hanawa, T.; Kodama, Y.; Boku, T.; Sato, M. “XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters”. In: Proceedings of the 1st Workshop on Accelerator Programming using Directives, 2014, pp. 27–36.

[NVI18] NVIDIA. “CUDA C Programming Guide”, 2018, Version PG-02829-001_v9.2.

[NVI19] NVIDIA. “Thrust | NVIDIA Developer”. Source: https://developer.nvidia.com/thrust, 12 nov 2020.

[Ope15a] OpenACC-Standard.org. “OpenACC Programming and Best Practices Guide”, 2015, Source: https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf.

[Ope15b] OpenMP Architecture Review Board. “OpenMP Application Programming Interface”, 2015, Version 4.5, Source: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[Ope17] OpenACC-Standard.org. “The OpenACC® Application Programming Interface”, 2017, Version 2.6, Source: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf.

[Ope18] OpenMP Architecture Review Board. “OpenMP Application Programming Interface”, 2018, Version 5.0, Source: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[Ora20] Oracle. “Package java.util.stream”. Source: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, 12 nov 2020.

[Pac11] Pacheco, P. S. “Introduction to Parallel Programming”. Morgan Kaufmann, 2011, 1 ed., 370p.

[PCR12] Pienaar, J. A.; Chakradhar, S.; Raghunathan, A. “Automatic generation of software pipelines for heterogeneous parallel systems”. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–12.

[Pie20] Pieper, R. L. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2020, 170p.

[Rei07] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”. O’Reilly, 2007, 1 ed., 336p.

[RGDF19] Rockenbach, D. A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “High-Level Stream Parallelism Abstractions with SPar Targeting GPUs”. In: Proceedings of the International Conference on Parallel Computing, 2019, pp. 543–552.

[RGF19] Rockenbach, D. A.; Griebler, D.; Fernandes, L. G. “Proposta de Suporte ao Paralelismo de GPU na SPar”. In: Anais de XIX Escola Regional de Alto Desempenho da Região Sul, 2019, pp. 4.

[RIK17] RIKEN AICS and University of Tsukuba. “XcalableACC Language Specification”, 2017, version 1.0, Source: https://xcalablemp.org/download/XACC/xacc-spec-1.0.pdf.

[RKBA+13] Ragan-Kelley, J.; Barnes, C.; Adams, A.; Paris, S.; Durand, F.; Amarasinghe, S. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”, *ACM SIGPLAN Notices*, vol. 46–6, Jun. 2013, pp. 519–530.

[RS01] Rumpf, M.; Strzodka, R. “Level set segmentation in graphics hardware”. In: Proceedings of the IEEE International Conference on Image Processing, 2001, pp. 1103–1106.

[RSG+19] Rockenbach, D. A.; Stein, C. M.; Griebler, D.; Mencagli, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “Stream Processing on Multi-Cores with GPUs: Parallel Programming Models’ Challenges”. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2019, pp. 834–841.

[SAE18] SAE International. “J3016 Recommended Practice: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles”, 2018.

[Sco12] Scott, S. “No Free Lunch for Intel MIC (or GPU’s)”. Source: https://blogs.nvidia.com/blog/2012/04/03/no-free-lunch-for-intel-mic-or-gpus/, 12 nov 2020.

[SHGW15] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K.-L. “Safe data parallelism for general streaming”, *IEEE Transactions on Computers*, vol. 64–2, Feb. 2015, pp. 504–517.

[SK10] Sanders, J.; Kandrot, E. “CUDA by Example: An Introduction to General-Purpose GPU Programming”. Addison-Wesley, 2010, 1 ed., 290p.

[SKG11] Steuwer, M.; Kegel, P.; Gorlatch, S. “SkelCL - A portable skeleton library for high-level GPU programming”. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 1176–1182.

[Spo02] Spolsky, J. “The Law of Leaky Abstractions”. Source: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/, 12 nov 2020.

[SRG19] Stein, C. M.; Rockenbach, D. A.; Griebler, D. “Paralelização do Dedup para Sistemas Multi-core com GPUs”. In: Anais de XIX Escola Regional de Alto Desempenho da Região Sul, 2019, pp. 363–366.

[SRG+20] Stein, C. M.; Rockenbach, D. A.; Griebler, D.; Torquati, M.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. “Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units”, *Concurrency and Computation: Practice and Experience*, May 2020, pp. e5786.

[SSB+19] Stein, C. M.; Stein, J. V.; Boz, L.; Rockenbach, D. A.; Griebler, D. "Mandelbrot Streaming para Sistemas Multi-core com GPUs". In: Anais de XIX Escola Regional de Alto Desempenho da Região Sul, 2019, pp. 339–342.

[Sta20] StarPU. "StarPU Handbook for StarPU 1.3.4". Inria, 2020, Source: http://starpu.gforge.inria.fr/files/doc/starpu.pdf.

[Str94] Stroustrup, B. "The Design and Evolution of C++". Addison-Wesley Professional, 1994, 1 ed., 480p.

[SU19] Shirke, S.; Udayakumar, R. "Lane Datasets for Lane Detection". In: Proceedings of the International Conference on Communication and Signal Processing, 2019, pp. 0792–0796.

[Szu16] Szuppe, J. "Boost.Compute: A parallel computing library for C++ based on OpenCL". In: Proceedings of the 4th International Workshop on OpenCL, 2016, pp. 1–39.

[TA10] Thies, W.; Amarasinghe, S. "An empirical characterization of stream programs and its implications for language and compiler design". In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 365–376.

[TAG+10] Turaga, D.; Andrade, H.; Gedik, B.; Venkatramani, C.; Verscheure, O.; Harris, J. D.; Cox, J.; Szewczyk, W.; Jones, P. "Design principles for developing stream processing applications", *Software: Practice and Experience*, vol. 40–12, Nov. 2010, pp. 1073–1104.

[TIO20] TIOBE. "TIOBE Index for July 2020". Source: https://www.tiobe.com/tiobe-index, 12 nov 2020.

[TKA02] Thies, W.; Karczmarek, M.; Amarasinghe, S. "StreamIt: A Language for Streaming Applications". In: Proceedings of the International Conference on Compiler Construction, 2002, pp. 179–196.

[Tor15] Torquati, M. "Parallel Programming Using FastFlow". Source: http://calvados.di.unipi.it/storage/tutorial/html/tutorial.html, 12 nov 2020.

[UGT09] Udupa, A.; Govindarajan, R.; Thazhuthaveetil, M. J. "Software pipelined execution of stream programs on GPUs". In: Proceedings of the 7th International Symposium on Code Generation and Optimization, 2009, pp. 200–209.

[VGF20] Vogel, A.; Griebler, D.; Fernandes, L. G. "Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multi-cores", *Software: Practice and Experience*, Dec. 2020.

[VGS+18] Vogel, A.; Griebler, D.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Autonomic and Latency-Aware Degree of Parallelism Management in SPar”. In: *Euro-Par 2018: Parallel Processing Workshops*, Springer, 2018, pp. 28–39.

[VJC+13] Verdoolaege, S.; Juega, J. C.; Cohen, A.; Gómez, J. I.; Tenllado, C.; Catthoor, F. “Polyhedral parallel code generation for CUDA”, *Transactions on Architecture and Code Optimization*, vol. 9–4, Jan. 2013, pp. 54:1–23.

[Vog18] Vogel, A. “Adaptive Degree of Parallelism for the SPar Runtime”, Master’s Thesis, School of Technology - Graduate Program in Computer Science - Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2018, 100p.

[VRJ+20] Vogel, A.; Rista, C.; Justo, G.; Ewald, E.; Griebler, D.; Mencagli, G.; Fernandes, L. G. “Parallel Stream Processing with MPI for Video Analytics and Data Visualization”. In: *High Performance Computing Systems*, Springer, 2020, vol. 1171, pp. 102–116.

[Whe16] Wheeler, D. A. “SLOCCount”. Source: https://dwheeler.com/sloccount/, 12 nov 2020.

[WTS04] Wang, Y.; Teoh, E. K.; Shen, D. “Lane detection and tracking using B-Snake”, *Image and Vision Computing*, vol. 22–4, Apr. 2004, pp. 269–280.

[Xca18] XcalableMP Specification Working Group. “XcalableMP Language Specification”, 2018, version 1.4, Source: https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf.

[ZM11] Zhang, Y.; Mueller, F. “GStream: A General-Purpose Data Streaming Framework on GPU Clusters”. In: Proceedings of the International Conference on Parallel Processing, 2011, pp. 245–254.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br