PUCRS

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

GABRIEL RUSTICK FIM

HIGH-LEVEL MULTI-GPU SUPPORT FOR MULTI-CORE STREAM PARALLELISM

Porto Alegre
2025

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 MULTI-GPU SUPPORT FOR MULTI-CORE STREAM PARALLELISM

GABRIEL RUSTICK FIM

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. Dr. Dalvan Griebler

Porto Alegre
2025

Ficha Catalográfica

R971h Rustick Fim, Gabriel

High-Level Multi-GPU Support for Multi-Core Stream Parallelism / Gabriel Rustick Fim. – 2025.
85.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Dalvan Jair Griebler.

1. Parallel Programming. 2. Data Parallelism. 3. Stream Processing. 4. Structured Parallel Programming. 5. GPU Programming. I. Griebler, Dalvan Jair. II. 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

GABRIEL RUSTICK FIM

HIGH-LEVEL MULTI-GPU SUPPORT FOR MULTI-CORE STREAM PARALLELISM

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 28th, 2025.

COMMITTEE MEMBERS:

Prof. Dr. Luiz Gustavo Leao Fernandes (PPGCC/PUCRS)

Prof. Dr. Claudio Schepke (LEA/UNIPAMPA)

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

PARALELISMO DE STREAM EM MULTI-GPU PARA MULTI-CORES

RESUMO

Atualmente, as arquiteturas de computadores dependem frequentemente de unidades de processamento gráfico (GPUs) para permitir a exploração massiva do paralelismo a um custo reduzido. Este paralelismo pode ser particularmente vantajoso no processamento de *streams*, um domínio de aplicações que processam continuamente um fluxo de dados de tamanho muitas vezes desconhecido. No entanto, o programador deve empregar programação paralela para explorar os recursos de hardware da GPU subjacente de forma eficiente. Isso pode ser desafiador, pois envolve refatorar algoritmos, usar técnicas de paralelismo e conhecer o hardware do ambiente, especialmente ao escrever código portável, uma vez que os fornecedores e gerações de GPU oferecem capacidades diferentes. Este desafio torna-se ainda mais complexo em ambientes multi-GPU; o programador deve escolher qual estratégia será utilizada para particionar seus dados, qual estratégia de escalonamento de tarefas será utilizada nas GPUs, como lidar com as necessidades de comunicação entre tarefas e como executar operações assíncronas na GPU. Para enfrentar esses desafios, pesquisadores se concentraram na investigação de técnicas de programação eficientes para GPUs e no desenvolvimento de abstrações que simplificam o processo de programação. Uma dessas abstrações é a SPar, uma linguagem de domínio específico (DSL) que permite a expressão do paralelismo de fluxo sem sacrificar o desempenho. Recentemente, foi adicionada uma extensão a SPar que permite a geração paralela de código para GPUs em aplicações de streaming. Para conseguir isso, a SPar realiza transformações de código fonte e gera código GPU usando uma biblioteca intermediária chamada GSParLib. No entanto, SPar oferece suporte à geração de código somente para ambientes com uma única GPU. Neste trabalho, investigamos como permitir a geração de código multi-GPU para processamento de *streams* e investigamos otimizações e técnicas para programação multi-GPU direcionado a sistemas multi-core. Nossas contribuições são um conjunto de algoritmos de escalonamento para fluxo de dados em multi-GPUs, que foram integrados na geração de código do SPar, suportando transpa-

rentemente o uso de multi-GPU em sistemas multi-core. Os resultados experimentais demonstraram que é possível simplificar a exploração de multi-GPU para aplicações de *stream* sem sacrificar o desempenho, utilizando políticas de escalonamento visando especificamente multi-GPU por meio de anotações de código como as fornecidas pelo SPar, alcançando resultados semelhantes às implementações manuais visando multi-GPU, enquanto tendo quase metade do número de linhas de código.

Palavras-Chave: Programação paralela, paralelismo de dados, processamento de *stream*, programação paralela estruturada, programação GPU, programação multi-GPU, linguagem específica de domínio, esqueletos algorítmicos, computação de alto desempenho, C, C++.

HIGH-LEVEL MULTI-GPU SUPPORT FOR MULTI-CORE STREAM PARALLELISM

ABSTRACT

Nowadays, computer architectures often rely on graphics processing units (GPUs) to allow massive parallelism exploitation at a lower cost. This parallelism can be particularly advantageous in stream processing, a domain of applications continuously processing a data flow of often unknown size. Nonetheless, the programmer must employ parallel programming to exploit underlying GPU hardware capabilities efficiently. This can be challenging since it involves refactoring algorithms, using parallelism techniques, and knowing about the environment's hardware, especially when writing portable code, since GPU vendors and generations offer different capabilities. This challenge becomes even more complex in multi-GPU environments; the programmer must choose which strategy to partition their data, which strategy to schedule their tasks onto the GPUs, how to handle communication needs between tasks, and how to perform GPU asynchronous operations. To address these challenges, researchers have focused on investigating efficient programming techniques for GPUs and developing abstractions that simplify the programming process. One such abstraction is SPar, a domain-specific language (DSL) that enables the expression of stream parallelism without sacrificing performance. Recently, an extension was added to SPar that allows parallel code generation for GPUs in streaming applications. To achieve this, SPar performs source-to-source code transformations and generates GPU code using an intermediate library named GSParLib. Nonetheless, SPar supports code generation for a single GPU environment only. In this work, we investigate how to allow multi-GPU code generation for stream processing and investigate state-of-the-art optimizations and techniques for multi-GPU programming targeting multi-core systems. Our contributions are a set of data stream scheduling algorithms for multi-GPUs, which were integrated in the code generation of SPar, transparently supporting multi-GPU usage in multi-core systems. The experimental results demonstrated that it is possible to simplify

the exploitation of multi-GPU for stream applications without sacrificing performance by utilizing scheduling policies specifically targeting multi-GPU through code annotations like the ones provided by SPar, achieving similar results to manual implementations targeting multi-GPU while having close to half the number of lines of code.

Keywords: Parallel programming, data parallelism, stream processing, structured parallel programming, GPU programming, multi-GPU programming, domain-specific language, algorithmic skeletons, high-performance computing, C, C++.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

API – Application Programming Interface

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

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DASP – Data Stream Processing

DSL – Domain-Specific Language

GCC – GNU Compiler Collection

GPU – Graphics Processing Unit

GSPARLIB – GPU Stream Parallelism Library

HPC – High-Performance Computing

MPI – Message Passing Interface

OPENACC – Open Accelerators

OPENCL – Open Computing Language

OPENMP – Open Multi-Processing

SIMT – Single Instruction Multiple Thread

SDK – Software Development Kit

SLI – Scalable Link Interface

SM – Stream Multiprocessors

SPAR – Stream Parallelism

CONTENTS

1. INTRODUCTION

Stream processing applications are commonly used to process a continuous flow of data, where the amount of data may not be known beforehand since they may be infinite streams. These applications are widely used in today's digital world, where data is generated from various sources such as social media, online shopping, and sensors from embedded systems [6]. To ensure optimal performance of stream processing applications, it is necessary to take advantage of the parallel capacity of the hardware on which the application is running. Programmers can achieve this by using a parallel programming model, which involves rewriting the serial code and breaking it down into smaller problems that can be solved concurrently [44].

To execute parallel programs more efficiently, programmers can offload each data parallel task of their workload to GPUs. This is because many stream processing applications can benefit from the massive parallelism offered by GPUs by implementing data parallelism inside a stage/operator[40, 55]. Even though GPUs began to gain popularity in the mid-1990s, via the high demand for high computing power to process 3D graphics spearheaded by the gaming industry launching games that utilized early 3D technology, their usage was extraordinarily convoluted. Because standard graphics APIs such as OpenGL and DirectX were still the only way to interact with a GPU, any attempt to perform arbitrary computations on a GPU would still be subject to programming constraints within a graphics API. Because of this, researchers explored general-purpose computation through graphics APIs by trying to make their problems appear to the GPU as traditional rendering [56].

According to [56] this changed in November 2006, when NVIDIA released its first DirectX platform and CUDA programming model. This model included several new components designed solely for GPU computing, aiming to overcome the limitations previously preventing graphics processors from being useful for general-purpose computation.

While GPUs have become ubiquitous in modern computing, coding for these architectures remains a complex task. It requires a thorough understanding of various programming techniques, depending on the supported GPU language, as well as familiarity with the hardware and concepts of many-core programming. Additionally, a deep understanding of the problem at hand is crucial, often necessitating the ability to leverage the massive parallelism that GPUs are renowned for.

This became an even more significant challenge when we began to tackle multi-GPU programming, as the architectures encompassed by this definition are highly diverse, varying from a simple one-node two-accelerator to a multi-node multi-accelerator per node cluster. As Sander and Kandrot [56] talk in their book, systems containing multiple graphics processors have become increasingly common in recent years. Products like

the GeForce GTX 295 include two GPUs on a single card. NVIDIA's Tesla S1070 contains four CUDA-capable graphics processors in it. Systems built around recent NVIDIA chipsets will have an integrated, CUDA-capable GPU on the motherboard, and adding a discrete NVIDIA GPU in one of the PCI Express slots will make this system multi-GPU.

Most, if not all, of the challenges of multi-GPU usage, revolve around how to efficiently utilize all the computational power provided by the multi-GPU environment; for this, one of the most critical points is to design inter-GPU communication properly since the efficiency of inter-GPU data transfers depends on how GPUs are connected within a node and across a cluster. Cheng, Grossman, and McKercher [12] say that there are two types of connectivity in multi-GPU systems: Multiple GPUs connected over the PCIe bus in a single node and multiple nodes containing GPUs connected over a network switch in a cluster, and both of these connections are not mutually exclusive.

Another challenge is efficiently partitioning the data between GPUs and splitting computation tasks into different work items. To Cheng, Grossman, and McKercher [12], for designing a program that takes advantage of multiple GPUs, you will need to partition the workload across devices, and depending on the application, this partitioning can result in two common inter-GPU communication patterns: No data exchange is necessary between partitions of a problem, and therefore no data shared across GPUs; or Partial data exchange between problem partitions, requiring redundant data storage across GPUs. To Wilt [65], since the GPUs can only use peer-to-peer to read or write data at PCIe rates, developers have to partition the workload in such a way that each GPU has about an equal amount of work to do, or the GPUs only need to interchange small amounts of data.

Another challenge of utilizing multi-GPU is how to schedule the usage of the GPUs present in the environment since it is of no use having multiple GPUs in a environment if they are being underutilized; knowing how to efficiently split the tasks that make a workload between the GPUs present in the environment is a must, but this challenge is not only limited to knowing how to efficiently divide the tasks between GPUs to utilize the environment entirely, it is intricately woven with the other challenges if the workload is going to be split between GPUs, then the data needed by the tasks will be needed to be partitioned. If the tasks build on each other or need to share some common data between works, there is also the need to manage these communications between GPUs.

There are some solutions present in the current literature that try to abstract some aspects of GPU programming, seeking to make the development of GPU-accelerated applications more feasible to the user by providing patterns or even more abstract directives and reducing the need for the user to fully engross themselves with the intrinsicalities of the language extensions, directives, or non-standard libraries that target GPUs. Although most of them have some support for multi-GPU, they still require challenging aspects of multi-GPU from the programmer, like data partitioning, scheduling, and com-

munication of GPUs, since these challenges cannot be so quickly abstracted since the multi-GPU environment has a varied spectrum of configurations.

One such solution is SPar [29, 30, 28]; it is a Domain Specific Language (DSL) embedded in C++ that offers high-level abstractions for stream parallelism through code annotations. The SPar compiler performs source-to-source transformations to generate parallel C++ code. The first version of SPar supported parallelism on multi-core architectures, making calls to the FastFlow [2]. In more recent studies, SPar was extended and enabled to generate TBB [33] and OpenMP [34] code for multi-core architectures, for clusters utilizing DSParLib (using MPI calls) [41, 49], and code targeting single-GPU using GSParLib (generating CUDA and OpenCL codes) [52, 53]. Showing promising results with comparable performance to manual implementations while requiring lower programming effort by the user [4, 3, 5].

Therefore, the main question that drives this research is: **Can C++ annotations, like those provided by SPar for stream parallelism, simplify multi-GPU parallelism exploitation without impacting performance?** To answer this question, we expect to provide the following scientific contributions:

- A methodology to allow multi-code generation through C++ attribute annotations;
- An extension of SPar language seeking to expand the GPU code generation for acceleration of stream processing to englobe multi-GPU usage;
- Comparative analysis of our proposed solution against state-of-the-art solutions for multi-GPU code generation.

This work is organized as follows: Chapter 2 presents the background for this study, including an overview of GPUs, state-of-the-art GPU programming languages, some challenges found when programming in a multi-GPU environment, GSParLib, and SPar. Chapter 3 presents the works that allow the programming and generation of multi-GPU codes, including frameworks based on structured parallel programming and code annotations. Chapter 4 presents the study of performance regarding GSParLib using manual implementations targeting multi-GPU. Chapter 5 presents the implementation of multi-GPU policies into SPar, as well as the study of performance of the multi-GPU implementation through the code abstractions provided by SPar. Chapter 6 presents our conclusions.

2. BACKGROUND

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 concepts of GPU and multi-GPU programming in Sections 2.2 and 2.3. In Sections 2.4, 2.5 and 2.6, we present language extensions that ease GPU programming tasks. Finally, Sections 2.8 and 2.7 present GSParLib and SPar, the latter being a DSL focused on stream parallelism and the focus of this study, that utilizes the first as an intermediary to generate its GPU code.

2.1 Stream Processing

A stream is a continuous flow of data [59], often generated by cameras, sensors, and other applications. The data that is streamed usually has business value that can only be realized through real-time processing and analysis. This results in strict performance requirements for the applications that process these streams, referred to as stream processing applications. Figure 2.1 lists some examples of stream processing applications.

Data Stream Processing (DaSP) is a computing paradigm that is represented by stream processing applications [24] [25]. These applications consume input data sources continuously and produce streams of output results [61]. As the world becomes more connected, digital data is produced at an ever-increasing pace, making these kinds of applications more common [61]. They are used in various domains, including data backup and compression, processing data from monitoring sensors and logs, financial markets, healthcare, and cryptography, among others.

Compared to traditional applications, stream processing does not have a defined end because the volume of data to be processed is usually unknown or unpredictable. In many cases, data flow comes from sensor measurements, and there are strict requirements for the latency and throughput of the data processing. It is not feasible to store the streamed data in a database and process them using traditional approaches [10].

In stream processing, each operator can handle a different data item from the previous operator. As a result, the degree of parallelism that can be achieved is typically limited by the number of operators present. However, any stateless operator can be duplicated to handle multiple data items simultaneously, further increasing the degree of parallelism that can be achieved.

The concept of providing algorithmic structures to simplify the process of parallel programming is not a new one. It has been explored under different names [28], such as algorithmic skeletons [13] [14] and parallel patterns [44]. The idea of defining patterns of

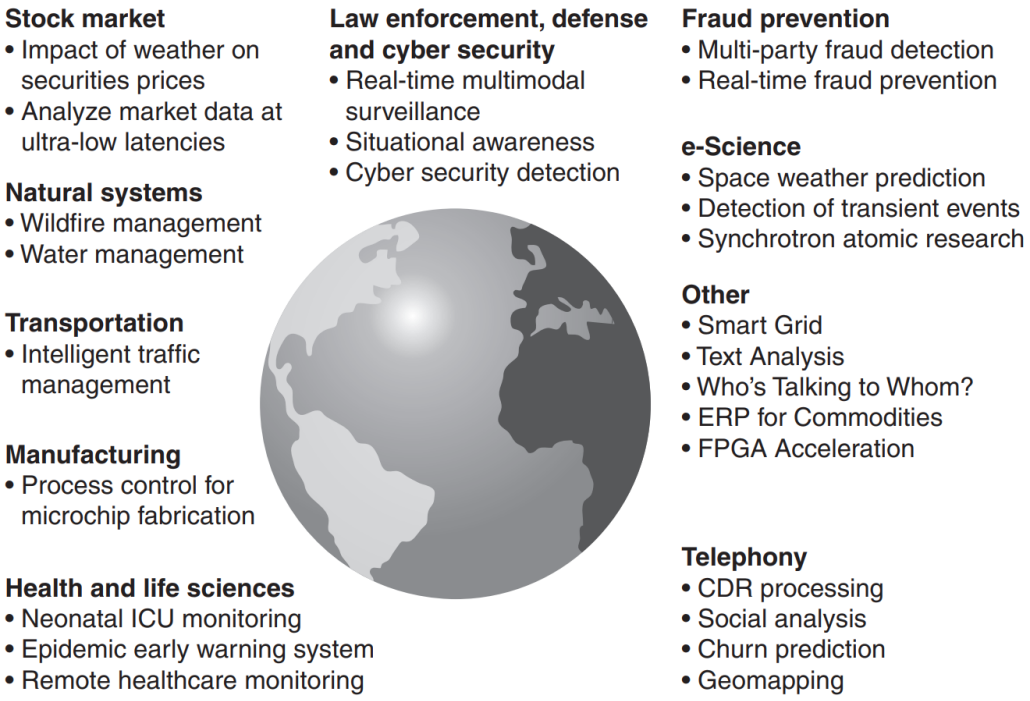

Figure 2.1 – Stream processing applications. Taken from [6]

commonly used programming tasks is derived from design patterns that are widely used in software engineering.

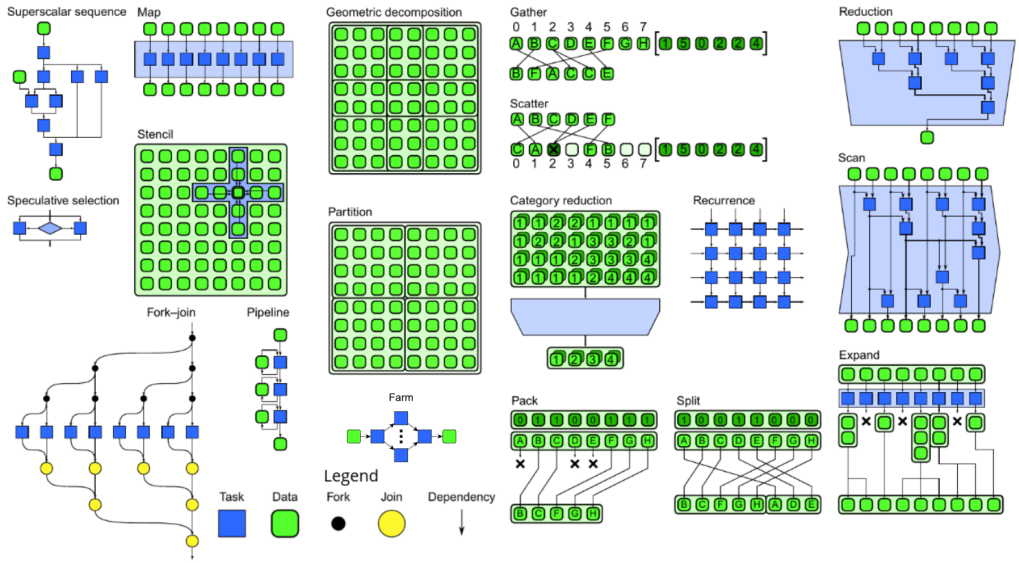

Figure 2.2 – An overview of parallel patterns. Taken from [44] [28] [52]

Figure 2.2 presents an overview with a visual representation of the main parallel programming patterns, being those:

- **Superscalar sequences**. As the name implies, this pattern defines a sequence of tasks that can be freely executed concurrently.

- **Speculative Selection**. In this pattern, both cases of a conditional statement run in parallel. When the conditions finish their execution, the unnecessary branch is canceled, and any data modification is reverted.
- **Map**. Applies the same computation over a set of data defined by an index. It is commonly used in a loop where the total of iterations is known, and each iteration is independent.
- **Stencil**. It is a variation of the map pattern. It accesses a data element and a set of neighbors. This pattern needs verification of bounds.
- **Fork-Join** A process creates a fork with other processes to compute other data portions. A process commonly waits for child processes to terminate its execution.
- **Pipeline**. A pipeline creates one stage for each operation, and a process or a thread executes each stage. Additionally, all stages are executed concurrently and must process each data element.
- **Geometric Decomposition**. It breaks the data into subsets that can overlap or not. Each subset is assigned to a thread or a process.
- **Partition**. It is a particular case of the geometric decomposition pattern where the subsets do not overlap.
- **Farm**. A farm has three stages: emitter, workers, and collector. The emitter produces data, which is processed by parallel workers in the second stage and sent to the collector in the third stage.
- **Gather**. It receives a set of indexes and reads the data only in the specified positions.
- **Scatter**. The inverse of the gather pattern, where the pattern writes the data instead of reading it.
- **Category Reduction**. Data is received and grouped utilizing a label before applying a reduction based on the label.
- **Pack**. It eliminates elements that are not being used in a set. Each element is marked with a Boolean; this indicates if the element is useful or not.
- **Split**. It is a variation of the pack pattern in which elements are moved to the leftmost or rightmost part of the arrays instead of removed.
- **Recurrence**. It is a generalization of loops where an iteration depends on another one.
- **Reduction**. Combines the values of a set of elements into a single value. It is commonly used to combine the results from different threads or processes.

- **Scan**. It is a variation of the reduction pattern; it computes every partial reduction from a set of elements.
- **Expand**. In the expand pattern, a map is executed, and each thread or process outputs zero or more elements. The result is a set with the outputs of every thread or process.

2.2 Graphics Processing Units

In the early days of computing, the CPU performed the calculations required for graphics applications, such as rendering 2D and 3D images, animations, and video. As more graphics-intensive applications were developed, their demands strained the CPU and decreased the computer's overall performance.

GPUs were developed to offload those tasks from CPUs for graphics applications. Some of the first graphics processing units (GPUs) were initially designed to process 2D and 3D graphics to support the game industry's demand.

This is reflected in the sense that the rapidly expanding video game industry has heavily influenced the design philosophy of GPUs. This is because advanced games require many floating-point calculations per video frame, which puts significant economic pressure on GPU vendors to maximize the chip area and power budget dedicated to these calculations. As a result, GPU vendor are always looking for ways to improve their products to meet the demands of the gaming industry [36].

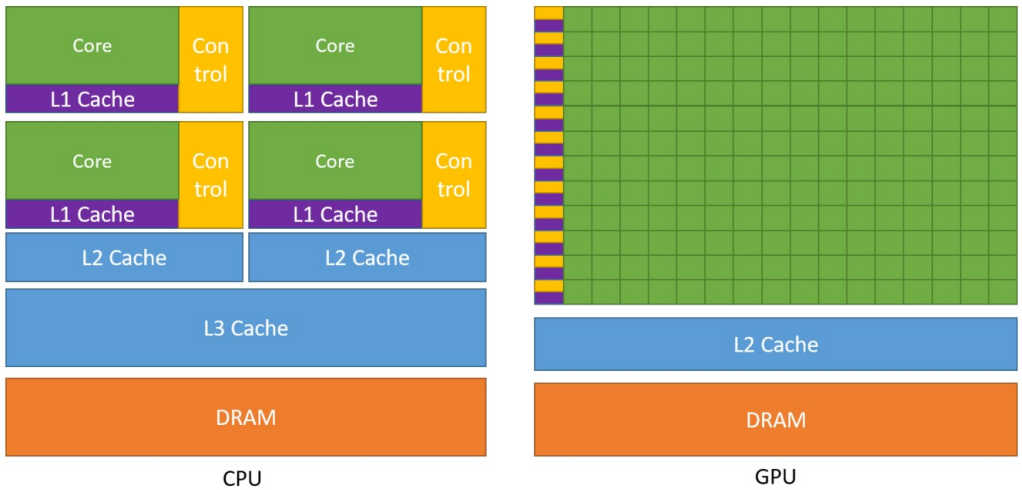

Figure 2.3 – CPU vs. GPU architecture. Taken from [21]

On the other hand, central processing units (CPUs) are engineered to expedite the execution of individual tasks. This is accomplished by leveraging on-chip caches to store frequently accessed data, reducing memory access time. Furthermore, the arithmetic

units and data delivery logic are optimized to expedite task completion, even if it requires greater power consumption and chip area.

The main reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and, therefore, designed such that more transistors are devoted to data processing rather than data caching and flow control [19].

More specifically, the GPU is especially well-suited to address problems expressed as data-parallel computations with high arithmetic intensity. Because the same program is executed for each data element, sophisticated flow control is less required. Because it is executed in many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

2.2.1 Thread Hierarchy

Current high-end GPUs have up to thousands of cores. They can run millions of threads, which are organized hierarchically. A function launched to the GPU is called kernel and is executed by a grid. A grid contains blocks, and each block has its threads. CUDA organizes its grids and blocks in three dimensions, allowing the number of blocks and threads to be configurable by the user [12]. Each thread on the block identifies its position, and each block identifies its position in the grid, allowing the thread to calculate its global ID by utilizing its position on the block and the block position on the grid [12].

Figure 2.4 presents an example of a grid of threads in CUDA. In the example, the grid has six blocks organized in two dimensions. Each bl ck has fifteen threads organized in three dimensions.

2.2.2 Memory Hierarchy

In general, applications do not access arbitrary data or run arbitrary code at any time. Instead, applications often follow the principle of locality, which suggests that they access a relatively small and localized portion of their address space at any time. Figure reffig:memHier shows an example of the memory hierarchy in CUDA; Shared Memory is visible to a block of threads. Global memory is visible to any thread of any grid.

Global memory is the largest, highest-latency, and most commonly used memory on a GPU. The name global refers to its scope and lifetime. Its state can be accessed on the device from any SM throughout the application’s lifetime. Shared memory enables threads within the same thread block to cooperate, facilitates the reuse of on-chip data,

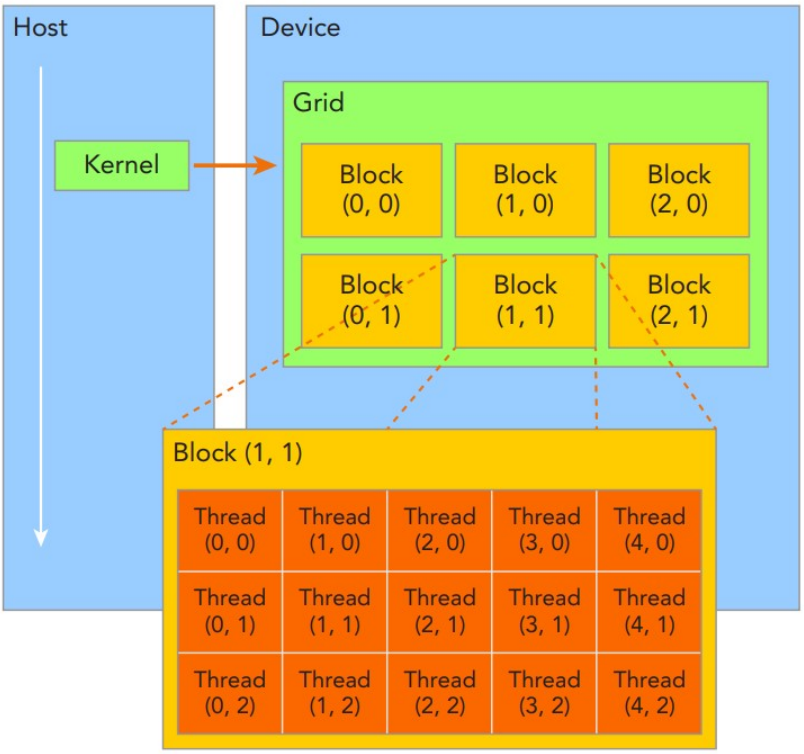

Figure 2.4 – CUDA Thread Hierarchy. Taken from [12]

and can significantly reduce the global memory bandwidth needed by kernels. Because the application explicitly manages the contents of shared memory, it is often described as a program-managed cache [12].

2.2.3 Execution Model

The GPU execution model is how the GPU executes the parallel code. Related to this concept, it is essential to understand GPU stream multiprocessors (SMs) and warps.

The GPU stream multiprocessors are responsible for executing thread blocks. When a rid of thread blocks is sent to the GPU, it gets distributed to the SMs. Then, the SMs execute the thread blocks concurrently. The more blocks created, the more blocks can be executed in parallel. However, each SM has a limited number of registers. If there are not enough registers available, the SM will execute fewer thread blocks in parallel [19].

In a computing system, warps serve as the fundamental unit of execution within an SM. When you initiate a grid of thread blocks, the thread blocks within the grid are distributed across the SMs. Once a thread block is assigned to an SM, threads within that block are further subdivided into warps. A warp comprises a set of 32 consecutive threads, and all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) mode.

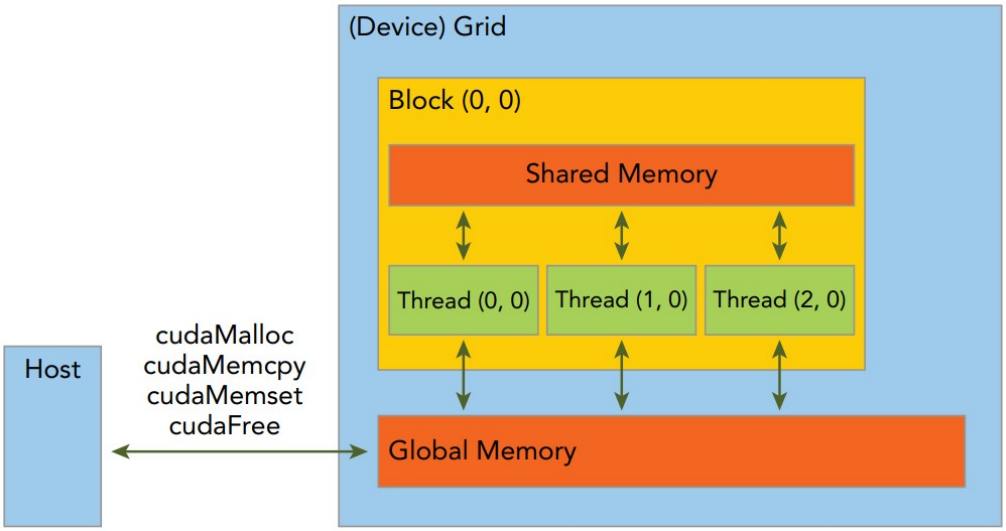

Figure 2.5 – CUDA Memory Hierarchy. Taken from [12]

This means all threads execute the same instruction, and each operates on its own private data [12].

2.2.4 Occupancy

GPU occupancy refers to the amount of parallel processing capacity a GPU uses. Fine-grained parallelism can be applied to explore a GPU's maximum capacity fully, which often requires restructuring the serial code. Launching multiple blocks of threads can also help increase GPU usage. Additionally, concurrent execution of GPU kernels is possible.

Instructions are executed sequentially within each core. When on warp stalls, the SM switches to executing other eligible warps. Ideally you want to have enough warps to keep the cores of the device occupied. Occupancy is the ratio of active warps to the maximum number of warps per SM [12].

When one thread requests to execute an instruction that takes multiple clock cycles to finish, another thread that is ready to run is scheduled, therefore, a GPU core does not remain idle.

2.3 Multi-GPU Programming

The CUDA API directly supports multiple GPUs, allowing the distribution of tasks between multiple GPUs. The API is low-level, utilizing it compared to using threads for multi-core utilization. Writing multi-GPU code this way requires careful manual orchestration of kernels and data movements and tends to be tedious and error-prone [43].

2.3.1 Data Partitioning

In data partition, different devices perform the same task concurrently on different parts of the input or output. There are two approaches to partitioning data: block partitioning and cyclic partitioning. In block partitioning, many consecutive elements of data are chunked together. Each chunk is assigned to a single thread in any order, and threads generally process only one chunk at a time. In cyclic partitioning, fewer data elements are chunked together. Neighbouring threads receive neighbouring chunks; each thread can handle more than one chunk. Selecting a new chunk for a thread to process implies jumping ahead as many chunks as there are threads.

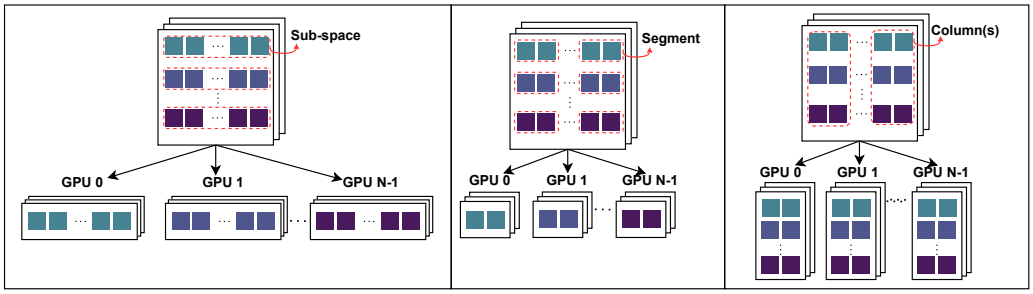

Figure 2.6 – Block Partitioning Strategies Examples. Based on [63]

Figure 2.6 shows three possible block partitioning strategies, all of which have their usage; GPU partitioning is particularly beneficial for workloads that do not thoroughly saturate the GPU's computing capacity. A lot of GPU workloads do not require a full GPU.

2.3.2 Multi-GPU communication

When developing in a multi-GPU environment the topic of how to handle the exchange of messages between GPUs is not trivial, mainly because traditionally, inter-GPU communication shares the same bus interconnect as CPU-GPU communication, such as PCIe. This changed recently due to the introduction of GPU-oriented interconnect such as NVLink, NVLink-SLI and NVSwitch. In this section we will briefly review some of this technology for multi-GPU communication.

- **PCIe**: The Peripheral Component Interconnect Express Bus (PCIe) is a standard for high-speed serial computer expansion. In a system with GPU integration, one or multiple GPU devices are connected to the CPUs through PCIe. However, when compared to the interconnect between CPU and DRAM, PCIe is significantly slower. This can often result in a major performance bottleneck for GPU-acceleration [48] [67].

- **NVLink**: NVLink is a communication interface that uses wires to connect nearby devices. It is based on High-Speed-Signaling-Interconnect (NVHS) [27] and supports peer-to-peer (P2P) communication, which allows linking between CPU-GPU or GPU-GPU. With NVLink, it is possible to directly read and write on the host memory of remote CPUs and/or the device memory of peer GPUs. Additionally, remote atomic operations can be performed. NVLink is bidirectional, which means that each link has two sublinks - one for each direction.

- **NVLink-SLI**: SLI [17] has traditionally been used for graphical purposes only [35]. However, the latest GPUs based on the Turing architecture have introduced a new form of high-speed multi-GPU bridge that utilizes NVLink interconnect technology. This bridge allows two GPUs to communicate with each other, enabling them to co-render games, co-run GPGPU tasks, or share GPU memory spaces.

- **NVSwitch**: NVSwitch [18] is designed to enable efficient all-to-all communication in deep neural network training and other emerging applications. It is an NVLink-based switch chip that features 18 ports of NVLink per switch for intra-node communication.

2.3.3 Multi-GPU Scheduling

Multi-GPU scheduling is a complex topic to summarize, so in this section, we will discuss some scheduling policies that could be used to allow the total usage of the GPUs that compose some environments. We will not talk about every single scheduling policy in the literature.

Predictable-Response-Time policy encourages any GPU command groups to wait for the completion of the preceding GPU command group, if any. Specifically, a new GPU command group arriving at the device driver can be submitted to the GPU immediately if the GPU work list is empty. Else, the corresponding task must sleep in the wait queue. The highest-priority task in the wait queue, if any, is woken up upon every interrupt from the GPU.

The High-Throughput policy reduces the scheduling overhead, compromising predictable response times a bit. It allows GPU command groups to be submitted to the GPU immediately if (i) the same task is submitted to the currently executing GPU command group and (ii) no higher-priority tasks are ready in the wait queue. Otherwise, they must be suspended in the same manner as the PRT policy. Upon an interrupt, the highest-priority task in the wait queue is woken up only when the GPU work list is empty.

OpenMP task-to-GPU scheduling strategy wherein OpenMP threads generate computational kernels on the CPU and then work together to dynamically map the tasks (each of which contains one or more kernels) to GPUs through a combination of application per-

formance tuning and runtime enhancements. The aim is that the loads are as balanced as possible, and the costs of the load balancing itself are also reduced through reducing dequeue and coordination overheads.

As the name implies, Centralized list scheduling with data locality is a scheduling policy based on a list of tasks. There is a GPU manager thread for each GPU in the system. If the GPUs are idle, the manager requests a task from a central scheduler, which initiates a data transfer of any data the prefetched task might need.

StarPU [7] approach the scheduling of tasks to multi-GPU via an abstract queue of tasks. Two operations can be performed on that queue: task submission (push) and request for a task to execute (pop). Several workers may share the actual queue provided its implementation protects it from concurrent accesses, thus making it transparent for the drivers.

Earliest-Finish-Time scheduling selects the task with the highest priority at each step. The chosen task is then assigned to the processor, which minimizes its earliest finish time with an insertion-based approach. A task's priority is the length of the critical path (i.e., the longest path) from the task to an exit task, including the computation cost of the task.

Work-stealing is a very well-known policy; each processor has a queue of work items to perform. Each work item consists of a series of instructions to be executed sequentially. Still, during its execution, a work item may also spawn new work items that can be executed in parallel with its other work. These new items are initially put on the queue of the processor executing the work item. When a processor runs out of work, it looks at the queues of the other processors and "steals" their work items.

2.3.4 Asynchronous Operations

In computer programming, asynchronous operation means that a process operates independently of other processes. In contrast, synchronous operation means the process runs only after another process is completed or handed off.

Asynchronous programming is a way to schedule work so that the GPUs can work concurrently with the tasks given. It is important to say that this does not mean that the GPU will necessarily run multiple kernels simultaneously. Often, asynchronous programming will mean overlapping memory transfer with kernel execution. This can improve efficiency since the GPU does not sit idle while transferring memory back and forth, resulting in improved throughput.

The general principle behind asynchronous operations on multi-GPU is to increase the overall unit throughput by reducing the number of unused warp slots and facilitating the simultaneous use of nonconflicting datapaths. The most basic communication setup

towards the GPU uses a single queue to synchronously push and execute graphics, compute, and copy workloads.

2.4 CUDA

In November 2006, NVIDIA unveiled the industry's first DirectX 10 GPU, the GeForce 8800 GTX. This was also the first GPU built with NVIDIA's CUDA Architecture[56]. This architecture was explicitly designed for GPU computing to overcome the limitations of previous graphics processors that could not be used for general-purpose computation.

CUDA is an extension to the C language that allows GPU code to be written in regular C. The code is either targeted at the host processor (CPU) or the device processor (GPU). The host processor spawns multithread tasks (or kernels as they are known in CUDA) onto the GPU device. The GPU has its internal scheduler that will allocate the kernels to whatever GPU hardware is present. Provided there is enough parallelism in the task, as the number of SMs in the GPU grows, so should the program's speed.

Code 2.1 presents a CUDA implementation for a vector summation. In the lines *2 - 7*, we define the GPU kernel that executes the vector summation. In line *3*, we calculate the thread's global id; since our example is straightforward, we are using a single dimension of the GPU, not needing to declare the formula to fully calculate the thread's id. The thread only calculates in line *4* if the if id corresponds to a valid position in the vectors.

```
1  #define N 1024
2  __global__ void addition(int *A, int *B, int *C){
3      int thread_id = threadIdx.x;
4      if (thread_id < N){
5          C[thread_id] = A[thread_id] + B[thread_id];
6      }
7  }
8  int main(){...
9      cudaMalloc(&a_dvc, N * sizeof(int));...
10     cudaMemcpy(a_dvc, a_host, N * sizeof(int), cudaMemcpyHostToDevice); ...
11     vecAdd<<<1, N>>>(a_dvc, b_dvc, c_dvc);
12     cudaMemcpy(c_host, c_dvc, N * sizeof(int), cudaMemcpyDeviceToHost); ...
13 }
```

Code 2.1 – CUDA Vector Summation.

In line *9*, we allocate the GPU memory for storing the vectors. In line *10*, we copy the vectors to the GPU. When programming CUDA, we must define the thread hierarchy by specifying the number of threads per block and the number of thread blocks in the grid. In this example, we create a GPU thread for each vector position.

In line *11*, we launch the GPU kernel to execute the vector summation algorithm. In line *12*, we copy the results from the GPU memory back to the host memory. To keep the examples simple, we decided to compress some declarations, such as allocations of

memory, since they are very similar, only exchanging the variables and native C++ code since they are not the focus of our examples.

NVIDIA created CUDA C by adding a small number of keywords to the industry-standard C language. This allowed them to utilize some unique features of the CUDA Architecture and make it more accessible to developers. A few months after launching the GeForce 8800 GTX, NVIDIA released a compiler for CUDA C. This made CUDA C the first language designed by a GPU company for general-purpose computing on GPUs.

2.5 OpenCL

OpenCL is an open and royalty-free standard supported by NVIDIA, AMD, and others. It sets out an open standard that allows using computing devices [15]. A computing device can be a GPU, CPU, or other specialist device for which an OpenCL driver exists. As of 2012, OpenCL supports all major brands of GPU devices, including CPUs with at least SSE3 support.

With OpenCL, you can write a single program that can run on a wide range of systems, from cellphones to laptops to nodes in massive supercomputers. No parallel programming standard has such a broad reach [45]. This is one reason why OpenCL is important and can transform the software industry, but it is also a source of criticism by some people.

Code 2.2 presents an OpenCL implementation for the matrix multiplication equivalent to the CUDA implementation from Code 2.1.

```
#include <CL/opencl.h>

const char *kernelSource =
" __kernel void addition ( __global int *A, ... ,const unsigned int n) { \n"
"   int thread_id = get_global_id (0) ; \n"
"   if (thread_id < n) { \n"
"       C[thread_id] = A[thread_id] + B[thread_id]; \n"
"   } \n"
"}";
int main(){
    unsigned int N = 1024;
    int a_host[N]; ...
    cl_mem a_dvc; ...
    cl_platform_id platform;
    cl_device_id device;
    cl_int status = clGetPlatformIDs(1, &platform, NULL);
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
    cl_command_queue queue = clCreateCommandQueueWithProperties(context, device, 0, &status);
    a_dvc = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) * N, NULL, &status); ...
    c_dvc = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int) * N, NULL, &status);
    status = clEnqueueWriteBuffer(queue, a_dvc, CL_FALSE, 0, sizeof(int) * N, a_host, 0, NULL, NULL); ...
    cl_program program = clCreateProgramWithSource(context, 1, (const char **)&kernelSource, NULL, &status);
    status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

```
25      cl_kernel kernel = clCreateKernel(program, "addition", &status);
26      status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dvc); ...
27      status = clSetKernelArg(kernel, 3, sizeof(unsigned int), &N);
28      size_t localSize = 64; size_t globalSize = N;
29      status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize, 0, NULL, NULL);
30      status = clEnqueueReadBuffer(queue, c_dvc, CL_TRUE, 0, sizeof(int) * N, c_host, 0, NULL, NULL); ...
31  }
```

Code 2.2 – OpenCL Vector Summation.

In the OpenCL example, in lines *3 - 9*, we are defining the computations of the OpenCL kernel. Like the CUDA example, the thread will only execute if its id corresponds to a valid vector position. In lines *13 - 17*, we create some of the variables related to the OpenCL API. In line *18* and *19*, we make the OpenCL context and command_queue. In lines *20 - 21*, we allocate the memory for storing the vectors and copy them in line *22*. In lines *23 - 25*, we create and build the OpenCL kernel using the string from lines *3 - 9*, lines *26 - 27* set the arguments for the OpenCL kernel. In the line *28*, we define the thread hierarchy; *local* is the number of threads per block, and *global* is the total amount of threads in the grid. In line *29*, we execute the OpenCL kernel. And in line *30*, copy the results from the GPU to the host.

[15] says that anyone familiar with CUDA can pick up OpenCL relatively quickly, as the fundamental concepts are similar. On the other hand, OpenCL is undeniably more complex than CUDA as it requires the programmer to perform many tasks automatically done by the CUDA runtime API.

This complexity is due to OpenCL delivering high portability levels by exposing the hardware, not hiding it behind abstractions like CUDA. This means the code with OpenCL must explicitly define the platform, its context, and how work is scheduled onto the devices.

2.6 OpenACC

The OpenACC Application Program Interface (API) describes a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs, and accelerators. OpenACC is similar to OpenMP regarding program annotation. Still, unlike OpenMP, which can only be accelerated on CPUs, OpenACC programs can be accelerated on a GPU or other accelerators also [8]. OpenACC aims to overcome the drawbacks of OpenMP by making parallel programming possible across heterogeneous devices. OpenACC standard describes directives and APIs to accelerate the applications.

OpenACC is a high-level, directive-based programming model for C/C++ and Fortran. It is designed to require significantly less programming effort than using a low-level model to program heterogeneous high-performance computing (HPC) hardware architectures [11].

Code 2.3 presents an OpenACC implementation for the matrix multiplications with a similar parallelism strategy adopted in the CUDA and OpenCL examples.

```
1  #define N 1024
2  void vecAdd(int *A, int *B, int *C){
3  #pragma acc kernels loop independent copyin(A[0:N], B[0:N]), copyout(C[0:N])
4      for (int i = 0; i < N; i++){
5          C[i] = A[i] + B[i];
6      }
7  }
8  int main(){
9      int a[N], b[N], c[N];
10     vecAdd(a, b, c); ...
11 }
```

Code 2.3 – OpenACC Vector Summation.

In the OpenACC example, we have a lower programming effort than CUDA and OpenCL. We apply the directive *#pragma acc kernels loop independent* on the vector summation algorithm, specifying that both the variables *A* and *B* are the inputs and that *C* is the output. Then, OpenACC creates a GPU thread for each position of the vectors. We do not need to specify memory transfers between the CPU and the GPU. OpenACC manages the memory automatically.

The OpenACC programming model is designed to help programmers parallelize their C/C++ or Fortran code without worrying about the complex details of the hardware platform. Programmers insert "hints" into their code to guide the compiler when compiling the code. This way, the compiler handles most of the translation details, leaving the programmer to concentrate on the code instead of the architecture.

OpenACC is a directive-based model that supplements a given codebase with "hints." This means the code can be compiled serially, disregarding the directives and producing accurate results. This approach maintains a single code base while providing portability across multiple platforms.

2.7 SPar

SPar is a domain-specific language (DSL) focused on expressing stream parallelism and was created by Dalvan Griebler in his PhD thesis [29, 28, 30]. The primary objectives of SPar are to optimize programmer productivity by eliminating the need for sequential code rewriting to exploit parallelism and to provide efficient programming abstractions that eliminate the need for the programmer to work on low-level or architecture-

dependent code. Recent studies have demonstrated the programmability benefits with beginners[4] and using coding metrics[3, 5].

In SPar, the parallelism is expressed using C++ attributes. Attributes are inserted between double square brackets and can be used to annotate types, classes, and code blocks, and may be put almost anywhere on the code [30]. SPar focuses on generating pipeline and farm parallel patterns, which are the parallel patterns best suited for stream parallelism. Initially, SPar focused on code generation for multi-core systems[34, 33], including the extension for defining service level objectives [32] and self-adaptive techniques[62]. Also, extension on the language for data parallelism were developed for shared-memory architectures [40, 39] and GPUs[54, 55], including GSParLib [52, 53] that enables code generation for GPUs using CUDA and OpenCL. For parallel distributed architectures, preliminary studies were carried out in [31] and later on [49] and with DSParLib [41] using MPI.

Code 2.4 shows an example of using SPar annotations to realize the vector summation with just data parallelism. In the example, the *for* loop reads, computes, and outputs the summation of the vectors.

```
1 [[spar::ToStream, spar::Input(size, arrayA[size], ...), spar::Output(size, arrayC[size])]]
2 for(int i = 0; i < MAX_SIZE; i++){
3     [[spar::Stage, spar::Pure, spar::Input(size, arrayA[size], ...), spar::Output(size, arrayC[size])]]
4     {
5         arrayC[i] = arrayA[i] + arrayB[i];
6     }
7 }
```

Code 2.4 – SPar Data Parallelism Vector Summation.

In line *1*, the ToStream annotation indicates that the following line is a stream region, and the stage is created. In this example, the first stage realizes the sum of the vectors *arrayA* and *arrayB* into the output vector *arrayC*.

Code 2.5 shows an example of combining stream and data parallelism in SPar to realize the sum of vectors. In line *1*, the ToStream annotation indicates that the following line is a stream region, and the stage is created. This first stage realizes the count of how many items the stream currently has.

```
1  [[spar::ToStream, spar::Input(item, size, arrayA[size], ...), spar::Output(item, size, arrayC[size])]]
2  for (int i = 0; i < MAX_STREAM_ITEMS; i++){
3      item = i;
4      arrayA = (int *)malloc(sizeof(int) * size);...
5      [[spar::Stage, spar::Input(item, size, arrayA[size], ...), spar::Output(item, size, arrayA[size], ...)]]{
6          item++;
7      }
8      [[spar::Stage, spar::Input(item, size, arrayA[size], ...), spar::Output(item, size, arrayC[size]), spar::
       Replicate()]]{
9          [[spar::Pure, spar::Input(item, size, arrayA[size], ...), spar::Output(item, size, arrayC[size])]]
10         for (int thd = 0; thd < MAX_ARRAY_SIZE; thd++){
11             arrayC[thd] = arrayA[thd] + arrayB[thd];
12         }
13     }
```

```
14      [[spar::Stage, spar::Input(item, size, arrayC[size]), spar::Output(item, size, arrayC[size])]]{
15          for (int thd = 0; thd < MAX_ARRAY_SIZE; thd++){
16              merge += arrayC[thd];
17          }
18      }
19  }
```

Code 2.5 – SPar Stream Processing Vector Summation.

In line *8*, the attributed *stage* is utilized to indicate the second stage of the algorithm. In this case, the second stage processes the elements, realizing the sum of the elements belonging to the stream. The third and last stage acts as a *reduce* stage, summing the results of the vector.

SPar uses C++ attributes as the mechanism of annotations to identify parallel regions in the serial code. Then, SPar generates the parallel code, making the necessary transformations in the serial code. SPar has five attributes defined for stream processing. Spar organizes the attributes into identifiers (ID) and auxiliary (AUX). Each SPar annotation inserted in the serial code must have an ID attribute and an optional list of AUX attributes.

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 an 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 SPar compiler was generated from CINCLE (Compiler Infrastructure for New C/C++ Language Extensions) [28]. CINCLE provides functionalities for C++ code analysis and an API for transformations in ASTs (Abstract Syntax Trees). Figure 2.7 illustrates the steps of the compilation process of the SPar compiler. The first step receives a C++ code annotated with C++ attributes. The second step performs a semantics analysis of the C++ code using GCC (GNU Compiler Collection). The third step scans the code and parses it to an AST. The fourth step applies the SPar transformation rules in the AST, generating a new AST. The fifth step compiles the source code transformed using GCC. The sixth step outputs a binary executable file of the parallel code generated.

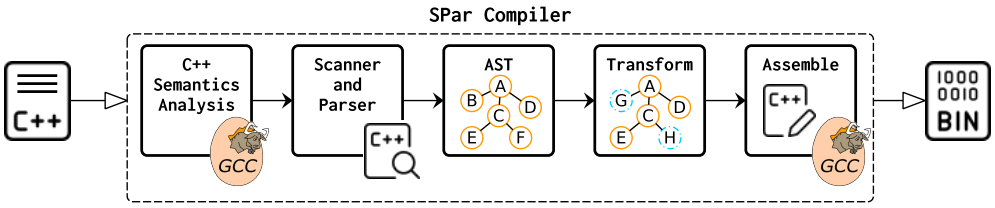

Figure 2.7 – Steps of the compilation process of SPar. Taken from [52]

The compiler supports three compilation flags that allow the programmer to control runtime behaviors in the stream processing. The flags are *spar_ondemand*, *spar_blocking*, and *spar_ordered*:

spar_ondemand uses the on-demand scheduler to control the flow of data. This scheduler keeps only one stream element at a time, which is processed by the stages of the stream pipeline. The default behavior is to continuously read stream elements and insert them into the communication queues between the stages of the stream pipeline.

spar_blocking uses blocking queues to communicate between the stages of the stream pipeline. Thus, only a single thread can access each communication queue. The default behavior is non-blocking queues. The performance of each type of queue can vary depending on the application.

spar_ordered sorts the stream's output data according to the input data's order. This functionality is critical when the order of the stream elements must be preserved.

Dinei's thesis [52, 55] adds two more flags to SPar, *spar_gpu* and *spar_opencl*; these flags are only used for the generation of GPU code:

spar_gpu activates the GPU extension of SPar, and SPar generates GSParLib source code when applicable. When using this flag, GSParLib uses the CUDA driver by default. *spar_opencl* makes GSParLib use the OpenCL driver when the GPU extension of SPar is activated.

2.8 GSParLib

GSParLib [52, 53] is a C++ library that aims to provide a C++ skeleton-based API, wrapping both CUDA and OpenCL languages. The library is divided into two APIs: a low-level API called Driver API and a high-level API called Pattern API. Driver API is a wrapper over CUDA and OpenCL languages, while the Pattern API comprises a set of parallel patterns built upon the Driver API.

Both APIs allow writing GPU codes without knowing the intricacies of writing CUDA or OpenCL codes by using a compilation flag; when using the flag GSPARDRIVER_CUDA, GSParLib compiles the code with CUDA, and when using GSPARDRIVER_OPENCL, GSParLib

compiles the code with OpenCL. Differently from these languages, on GSParLib, the GPU kernels must be defined as a string inside the code; this allows GSParLib to modify the kernel source code to better suit the hardware characteristics collected during execution.

Code 2.6 presents the vector summation utilizing the Driver API. In lines *1 - 7* we define the compiling language that will be used throught the compilations flags provided by GSParLib. Lines *15 - 19* declare the kernel code, following a similar implementation to CUDA and OpenCL. Lines *20 - 30*, we allocate the GPU memory and compile the kernel code together with any additional routine, such as the one present in line *13*. Lines *31 - 39* sets the variables the kernel utilizes and runs the GPU kernel.

```
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  int *array_1; ...
9  Instance* driver;
10 MemoryObject* array_1_dvc; ...
11 Kernel* array_summation_kernel;
12 extern std::string source_array_summation_kernel;
13 std::string source_additional_routines = "#define SIZE 1024\n";
14
15 std::string  source_array_summation_kernel = GSPAR_STRINGIZE_SOURCE(
16 __gspar_device_kernel__ void array_summation_gpu(
17   __gspar_device_global_memory__ int* array_1, ...){   int thread_id = gspar_get_global_id(0);
18   if(thread_id >= SIZE){return;}
19   array_3[thread_id] = array_1[thread_id] + array_2[thread_id];});
20 void initialization(int *array_1, int *array_2, int *array_3){ ...
21   driver = Instance::getInstance();
22   driver->init();
23   int numGpus = driver->getGpuCount();
24   if (numGpus == 0) {... exit(-1);}
25   auto gpus = driver->getGpuList();
26   auto gpu = driver->getGpu(0);
27   array_1_dvc = gpu->malloc(sizeof(int *) * SIZE, array_1); ...
28   try{ std::string complete_kernel_source = "";   complete_kernel_source.append(source_additional_routines);
         complete_kernel_source.append(source_array_summation_kernel);
29     array_summation_kernel = new Kernel(gpu, complete_kernel_source, "array_summation_gpu");
30   } catch (GSPar::GSParException &ex) {... exit(-1);}}
31 void array_summation(){
32   int threads = 64; int blocks = SIZE;
33   try { array_summation_kernel->clearParameters();
34     array_summation_kernel->setNumThreadsPerBlockForX(threads);
35     array_summation_kernel->setParameter(array_1_dvc); ...
36     unsigned long dimensions[3] = {(unsigned long)(blocks), 0, 0};
37     array_summation_kernel->runAsync(dimensions);
38     array_summation_kernel->waitAsync();
39   } catch (GSPar::GSParException &ex) {...}}
40 int main(int argc, char const *argv[]){
41   array_1 = (int *)malloc(sizeof(int *) * SIZE); ...
42   initialization(array_1, array_2, array_3);
43   array_1_dvc->copyIn(); ...
44   array_summation();
45   array_3_dvc->copyOut(); ... }
```

Code 2.6 – GSParLib Driver API Vector Summation.

The Driver API abstracts most concepts shared between the CUDA and OpenCL languages from the programmer, while some specific concepts are wrapped in helper classes to ease the programming. The Driver API class and methods have simple names directly related to the domain of GPU programming, which anyone with some GPU programming knowledge should recognize.

Code 2.7 presents the vector summation utilizing the Pattern API. It has very similar code to the example using the Driver API, the significant difference being that the kernel declaration only contains the code that needs to be executed. Another difference is how to build the kernel and call its execution.

```
#include "GSPar_PatternMap.hpp"
#include "GSPar_PatternReduce.hpp"
#include "GSPar_PatternComposition.hpp"
using namespace GSPar::Pattern;
#ifdef GSPARDRIVER_CUDA
#include "GSPar_CUDA.hpp"
using namespace GSPar::Driver::CUDA;
#else
#include "GSPar_OpenCL.hpp"
using namespace GSPar::Driver::OpenCL;
#endif
int *array_1; ...
Instance* driver;
MemoryObject* array_1_dvc; ...
Map* array_summation_kernel;
extern std::string source_array_summation_kernel;
std::string source_additional_routines = "#define SIZE 1024\n";

std::string  source_array_summation_kernel = GSPAR_STRINGIZE_SOURCE(
  int thread_id = gspar_get_global_id(0);
  if(thread_id >= SIZE){return;}
  array_3[thread_id] = array_1[thread_id] + array_2[thread_id];
);
void initialization(int *array_1, int *array_2, int *array_3) { ...
  driver = Instance::getInstance();
  driver->init();
  int numGpus = driver->getGpuCount();
  if (numGpus == 0) { std::cout << "No GPU found, interrupting the benchmark" << std::endl; exit(-1);}
  auto gpus = driver->getGpuList();
  auto gpu = driver->getGpu(0);
  array_1_dvc = gpu->malloc(sizeof(int *) * SIZE, array_1); ...
  array_1_dvc->copyIn(); ...
  int threads = 64; int blocks = SIZE;
  try { unsigned long dimensions[3] = {(unsigned long)(blocks), 0, 0};
    array_summation_kernel = new Map(source_array_summation_kernel);
    array_summation_kernel->setStdVarNames({"gspar_thread_id"});
    array_summation_kernel->setParameter<int*>("array_1", array_1_dvc, GSPAR_PARAM_PRESENT); ...
    array_summation_kernel->setNumThreadsPerBlockForX(threads);
    array_summation_kernel->addExtraKernelCode(source_additional_routines);
    array_summation_kernel->compile<Instance>(dimensions);
  } catch (GSPar::GSParException &ex) { ... exit(-1); } }
void array_summation() {
```

```
43      array_summation_kernel->setParameter<int*>("array_1", array_1_dvc, GSPAR_PARAM_PRESENT); ...
44      array_summation_kernel->run<Instance>();}
45  int main(int argc, char const *argv[]){
46    array_1 = (int *)malloc(sizeof(int *) * SIZE); ...
47    initialization(array_1, array_2, array_3);
48    array_summation();
49    array_3_dvc->copyOut(); ...
50  }
```

Code 2.7 – GSParLib Pattern API Vector Summation.

The Pattern API builds upon the Driver API. It offers the programmer a set of parallel patterns, with the Map and Reduce patterns already created as classes. The programmer can also write their patterns for later use.

3. RELATED WORK

In this chapter, we present and analyze related research. For this purpose, we consider related scientific documents that addressed programming in a multi-GPU environment, be it utilizing structured parallel programming frameworks or frameworks that generate GPU code through annotations on the host code. As well as works that touch on the different techniques for multi-GPU programming.

3.1 Structured Parallel Programming with multi-GPU

This section briefly presents GPU frameworks based on wrappers and parallel patterns closely related to GSParLib, which SPar uses to generate GPU code for stream processing acceleration.

SkePU [26] is a C++ library based on parallel patterns. It provides the parallel patterns Map, Reduce, MapReduce, Stencil, Scan, MapOverlap, MapPairs, and MapPairsReduce. To apply parallelism using the available parallel patterns, a programmer must use C++ templates. SkePU is a tool that can generate OpenMP code for CPUs or CUDA and OpenCL for GPUs. Unlike GSParLib, SkePU uses smart data containers and custom data structures resident in the host memory. The smart data containers automatically manage the memory device using accelerators like GPUs. These containers identify when to allocate memory or transfer data. However, one drawback of this approach is that it provides a non-native C++ data structure. Another drawback is when the algorithm behind the smart data containers does not choose the optimal method to optimize the memory transfers.

SkeCL [58] is a C++ framework that generates OpenCL code. It provides Map, Reduce, Zip, Scan, and Stencil patterns. To utilize parallel patterns, the programmer needs to create a class containing the desired pattern and then use a string parameter containing the code to be executed in parallel. SkelCL will generate the low-level OpenCL code automatically. The approach used by SkelCL is quite similar to that of GSParLib.

Musket is an alternative to the Muesli library; in [66], the authors expand Musket to generate C++ code and additional project files for multi-core and multi-GPU environments in CUDA and OpenACC. Differently from SPar, which is an internal DSL for annotating a sequential program, which is then transformed into a FastFlow application, their implementation provides an external DSL, reducing the complexity since there is no need for annotations or pragmas to be embedded within another language.

MGPU [57] is a C++ programming library aimed at single-node multi-GPU environments depending on a few Boost C++ libraries. It works as a layer on top of exist-

ing GPU computing frameworks and numerical libraries, supporting frameworks such as CUDA and OpenCL, focusing more on CUDA. Application built with MGPU can detect various performance-relevant architecture features, such as the number of devices present and the capabilities of each device. This allows MGPU to enable optimized versions of functions such as peer-to-peer for inter-GPU communication, which is much faster than transfers staged through the host.

MultiSkel [37] is a library built upon the CUDA framework and pthreads library; the library plays a role at the abstract level, taking away the need to pay attention to details of the model of architecture or implementation of CUDA or how to make multi-threads to control multi-GPUs.

dOCAL [50] is a C++ library that combines major advantages over the state-of-the-art approaches, simplifying implementation of both OpenCL and CUDA codes by automatically managing low-level details such as data transfers and synchronization, executing user-provided OpenCL and CUDA kernels; it enables conveniently targeting the devices of multi-node systems by automatically managing the node-to-node network communication; it simplifies data transfer optimizations by providing different, specially allocated memory classes, pinned main memory for overlapping data transfers with computations; it optimizes memory management by automatically detecting and avoiding unnecessary data transfers; it enables interoperability between OpenCL and CUDA host code by automatically handling the communication between OpenCL and CUDA data structures and by automatically translating between the OpenCL and CUDA kernel programming languages.

HIP [16] is a C++ API that allows the programmer to use a single source code that can run on AMD and NVIDIA GPUs. The HIP programming language can generate code for AMD GPUs using Radeon Open Compute (ROCm) and NVIDIA GPUs using CUDA. ROCm, like CUDA, is a platform designed specifically for AMD GPUs. It acts as a wrapper around the GPU's mechanisms and provides a unified interface for both GPU backends. It also doesn't have any noticeable overhead compared to CUDA.

Kokkos [60] is an API written in C++, which allows code portability across various high-performance computing platforms. Currently, Kokkos supports several backends, such as CUDA, HIP, SYCL, HPX, OpenMP, OpenMPTarget, and C++ threads, providing parallel execution and data management abstractions. This means that when using Kokkos, a single source code can be used on different HPC platforms. However, the programmer must manually provide CUDA or HIP code when targeting a particular architecture, such as a GPU.

SYCL [51] is a programming model designed for heterogeneous systems that enables using different devices, such as CPUs, GPUs, and FPGAs, in a single application. It is an open standard managed by the Khronos Group and utilizes a single-source embedded domain-specific language based on pure C++17. The primary aim of SYCL is to provide a

consistent language, APIs, and ecosystem for writing and optimizing code for accelerator architectures while allowing optimized kernel code to vary across different architectures.

FastFlow [1] is a C++ framework that supports stream and data parallelism based on parallel patterns. It provides several parallel patterns for CPUs, including pipeline, farm, map, mapReduce, and stencil. For GPUs, a single parallel pattern called loop-of-stencil-reduce can implement the patterns map, reduce, and stencil.

NVIDIA HPC SDK [22] is a Software Development Kit created by NVIDIA to maximize developer productivity and the performance and portability of HPC applications. One of its compilers, NVC++ [20], allows the compilation of GPU-accelerated Standard C++ without language extensions, directives, pragmas, or non-standard libraries. This allows any code written to be easily ported between systems and automatically accelerated with high-performance NVIDIA GPUs. CUDA Unified Memory implicitly and automatically controls all data movement between host and GPU device memory.

Table 3.1 presents general information about the related work compared to GSParLib. Column *GPU Backend* indicates which backend is supported for each framework; for example, FastFlow supports both CUDA and OpenCL. Column *Backend Abstraction* indicates if the framework requires low-level code from the backend; for example, both Kokkos and FastFlow require that the GPU kernels be written utilizing the desired backend syntax. Column *Supported Patterns* lists all the GPU parallel patterns supported by the framework. Column *Multi-GPU Support* indicates if the framework supports the generation of multi-GPU code. Column *Stream Accessible* indicates if the GPU mechanisms required for programming stream applications are transparent to the user or facilitated through high-level abstractions.

SkePU, dOCAL, FastFlow, and GSParLib support both CUDA and OpenCL. CUDA is the standard form for exploiting parallelism on NVIDIA GPUs, the prominent GPU-based environments. However, CUDA only supports NVIDIA GPUs. On the other hand, OpenCL offers fewer resources than CUDA but permits exploiting parallelism on GPUs from different vendors and even other accelerators. Thus, supporting both backends is essential for the code's best performance and portability.

GSParLib offers fewer parallel patterns than most related work because GSParLib is still an initial project. Still, the parallel patterns already supported by it provide considerable flexibility to approach different problems if used to the fullest.

All related works allow the generation of multi-GPU code through their framework. Still, most of them are straightforward implementations, leaving the bulk of the challenge of multi-GPU usage to the user since the environment variances are too significant to be entirely abstracted. In contrast, most related works don't provide an accessible stream abstraction for stream processing and GPU processing. GSParLib automatically provides mutual exclusion sessions for CPU threads when the programmer manipulates critical GPU resources, such as allocating or freeing memory. Also, GSParLib automatically

Table 3.1 – Frameworks based on Structured Parallel Programming

Ref	Name	GPU Backend	Backend Abstraction	Supported Patterns	Multi-GPU Support	Stream Accessible
[26]	SkePU	CUDA, OpenCL	Yes	Map, Reduce, Scan, MapReduce, MapOverlap, MapPairs–Reduce, Call	Yes	No
[58]	SkeCL	OpenCL	Yes	Gather, Map, Reduce, Scan, Stencil	Yes	No
[66]	Musket	CUDA, OpenACC	Yes	Fold, MapFold, Reduce, MapReduce	Yes	No
[57]	MPGU	CUDA	Yes	Map, Reduce	Yes	No
[37]	MultiSkel	CUDA	Yes	Map, Reduce, ZipWith, Scan, MapReduce, ZipWithReduce	Yes	No
[50]	dOCAL	CUDA, OpenCL	Yes	Not Available	Yes	No
[16]	HIP	CUDA, ROCm	Yes	Not Available	Yes	No
[60]	Kokkos	CUDA, HIP, SYCL	No	For, Reduce, Scan	Yes	No
[51]	SYCL	OpenCL	Yes	Not Available	Yes	No
[1]	FastFlow	CUDA, OpenCL	No	Map, Reduce, Stencil	No	No
[22]	HPC SDK	CUDA, OpenMP, OpenACC	Yes	Not Available	Yes	No
[52]	GSParLib	CUDA, OpenCL	Yes	Map, Reduce	Yes	Yes

creates CUDA streams and OpenCL queues and manages them to allow asynchronous GPU kernels. The related work commonly only provides wrappers over CUDA streams and OpenCL command queues. Consequently, the user must manually implement and manage all the GPU mechanisms required for stream processing, imposing a significant programming effort.

3.2 Annotation-based Programming with multi-GPU

This section presents frameworks based on code annotations, similar to SPar, that allow the generation of code targeting multi-GPU environments.

OpenMP [9] is a standard for multi-core CPUs and allows parallelism in code through directives. From version 4.0, OpenMP supports generating code for GPUs. It enables the usage of multi-GPU through a multi-thread approach, where each thread controls one GPU or one node containing one or more GPUs.

OpenACC [47] is a framework that utilizes code annotations to generate GPU code. Like OpenMP, the basic directives are available on OpenACC; however, OpenACC has a more extensive set of directives and functionalities.

XACC [46] is an extension of the XMP framework that focuses on generating GPU code using OpenACC instead of CUDA. Although XACC and CUDA are similar in improving

programmability, they still require knowledge of GPU and distributed programming details, such as memory transfers between CPU and GPU or nodes.

MACC [42] is a transpiler that extends the OpenACC programming model to allow applications to be seamlessly used in multi-GPU Environments. Their approach is a source-to-source transformation from the OpenACC code into post-source code that exploits both OpenACC and OpenMP.

Table 3.1 presents general information about the related work compared to SPar. Column *Annotation Method* indicates the mechanism used by the framework to offer the functionality of annotations. Column *GPU Backend* indicates which backend is supported for each framework. Column *Backend Abstraction* indicates if the framework requires low-level code from the backend. Column *Multi-GPU Support* indicates if the framework supports the generation of multi-GPU code. Column *Stream Paralellism* lists the capacity of simultaneous parallelism offered by the framework.

Table 3.2 – Frameworks based on code annotations

Ref	**Name**	**Annotation Method**	**GPU Backend**	**Backend Abstraction**	**Multi-GPU Support**	**Stream Paralellism**
[9]	OpenMP	Pragma	Compiler dependent	Yes	Yes	CPU, GPU
[47]	OpenACC	Pragma	Compiler dependent	Yes	Yes	GPU
[46]	XACC	Pragma	OpenACC	Yes	Yes	GPU, Distributed
[42]	MACC	Pragma	OpenACC	Yes	Yes	GPU, Distributed
[28]	SPar	C++ Attributes	GSParLib	Yes	No	CPU, GPU, Distributed

Although all of the frameworks present in Table 3.2 offer portability between GPUs of different vendors by being able to generate CUDA and OpenCL code. Only SPar offers the opportunity to explore three levels of parallelism: CPU, GPU, and Distributed. It is also the only one not to use pragma as the annotation method, utilizing C++ Attributes. The user provides the source code annotated with C++ attributes. Then, the compiler generates parallel code for the CPU, GPU, or cluster without requiring any other low-level mechanisms.

In comparison to related work, SPar requires significantly less programming effort. It uniquely abstracts stream parallelism on the CPU and data parallelism on the GPU, providing portability for various GPU vendors. However, SPar has certain limitations. OpenMP and OpenACC directives optimize the serial code before applying GPU parallelism. Unfortunately, SPar does not currently offer attributes for GPU optimization, so programmers have to manually apply transformations to the serial code before annotating the C++ attributes.

4. MULTI-GPU RUNTIME SUPPORT

In this chapter, we briefly introduce the initial policies planned to be implemented into SPar to multi-GPU scheduling Section 4.1), what were the chosen applications utilized to test the performance of the policies (Section 4.2), and some fine-tuning tests done utilizing the policies in chosen applications (Section 4.3).

4.1 Multi-GPU Scheduling policies

This section presents the multi-GPU scheduling policies tested in our experiments. To choose the policies that we planned to implement into SPar, we sought policies currently used when working with multi-GPU in the literature and common approaches to work division between multiple processors, in the end, we chose to test four policies, two focusing on scheduling techniques and two focusing on load-balancing between processors.

We can divide the policies tested into three main variations: one based on fully saturating a GPU with work before electing to send it to another, a static policy where each processing thread has access to a designated GPU, and a queue approach where all work is stored in a queue so that each GPU can take a work to be done.

The Static approach, shown in Figure 4.1, sends works to each processing thread spawned; these threads each have a set GPU linked to them, and the linked GPU is based on the remainder of the processing thread ID divided by the number of GPUs in the environment, this approach will work as our comparison base since it is simple, easy to implement, and starvation-free.

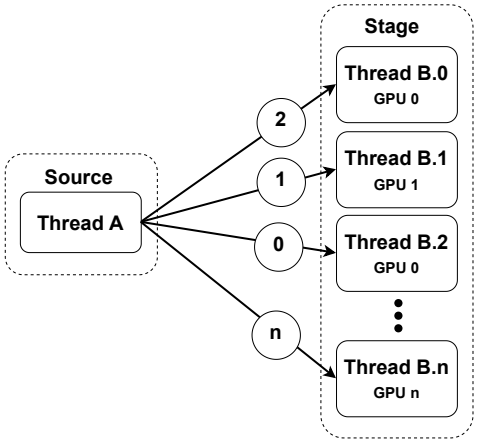

Figure 4.1 – Static representation

Saturate, shown in Figure 4.2, seeks to fill a GPU with work before sending any work to the following GPU, in short it tries to achieve a 100% usage of the GPU while also trying to not overfill the available memory of the targeted GPU. Since our test environment was comprised of a machine with 2 GPUs, we divided this policy into two branches, one targeting the stronger GPU on the environment, aliased as *Strong* in the later graphs, and other targeting the weaker GPU, denoted as *Weak* in the tests. On this policy, the threads do not have a set GPU for executing their processing; instead, when generating the work the source operator query the current usage and available memory of the GPUs and uses this information to decide to which GPU the work is going to be sent, sending the chosen GPU ID together with the work needed to processed.

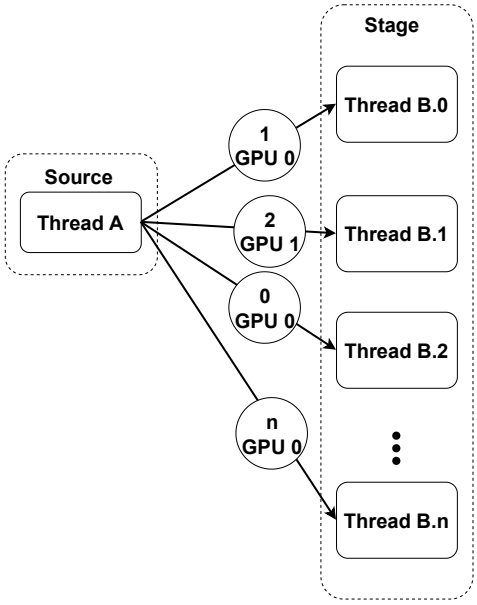

Figure 4.2 – Saturate representation

The Queue approach shown in Figure 4.3 was created due to its similarities to FastFlow's load-balancing approach, giving early insight into how the policies could behave when ported to SPar, but differently from FastFlow's queue per worker thread. In contrast, it has the same idea present on StarPU's work[7] in creating a queue for each GPU present in the environment. This approach is easily modified to introduce the usage of Work Stealing, shown in Figure 4.4. Between the GPUs, where if the queue belonging to its GPU is empty, it can take a task from the other queues. It follows the same approach as the Static, with a set GPU for usage in each thread that is spawned.

While there are other scheduling policies, such as priority scheduling or shortest job first, that could be used, after some deliberation, we reached a consensus that they are not feasible to be implemented for SPar without changing the underlying FastFlow execution structure or do not provide an explicit usage when targeting the usage of multi-GPU for streaming.

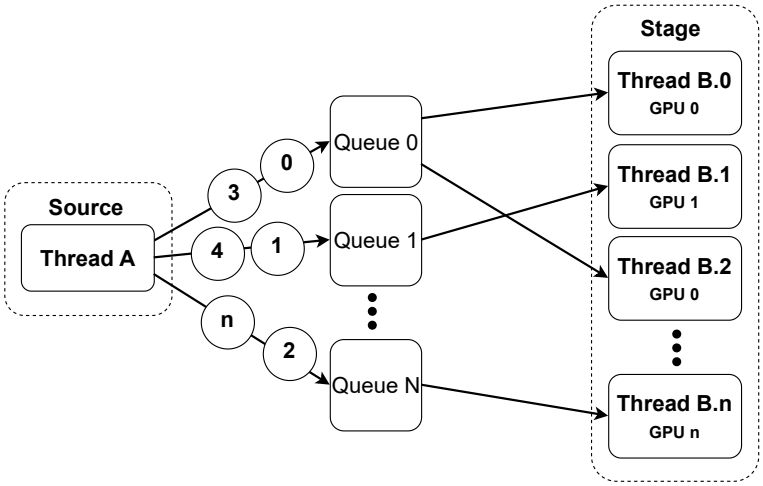

Figure 4.3 – Queue representation

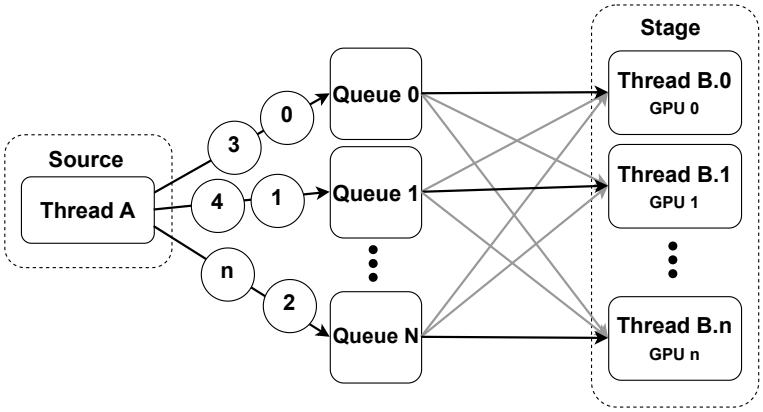

Figure 4.4 – Work Stealing representation

4.2 Chosen applications

In this Section, we describe the applications chosen to be tested using the policies, the choice of this applications was based on the fact that they were previously utilized to evaluate GSParLib's and SPar's codes targeting GPU environments. AnimalRescue (AR) is a reduced version of the application shown in the work of [23]. It is a synthetic stream processing application that simulates a continuous stream of data collected from drones. The application is composed of heavy computations and allows data parallelism to be exploited in each stream element. Each element contains a list of coordinates captured by a drone and a list of animals; the objective of each drone is to find the best coordinate for each animal from its list. Each animal has a different type and requirements for a proper localization.

Figure 4.5 shows the flowchart of the AnimalRescue application. According to [23], the application could be replicated in stages B, C, and D because they are stateless.

Both stages A and E are operations of IO, meaning that they read the inputted data or write the processed data to the disk. Stage B is responsible for extracting information from each drone's coordinates, such as average height. Stage C is responsible for finding the best coordinate for each animal utilizing the extracted data from Stage B. Stage D's objective is to validate the results found in Stage C.

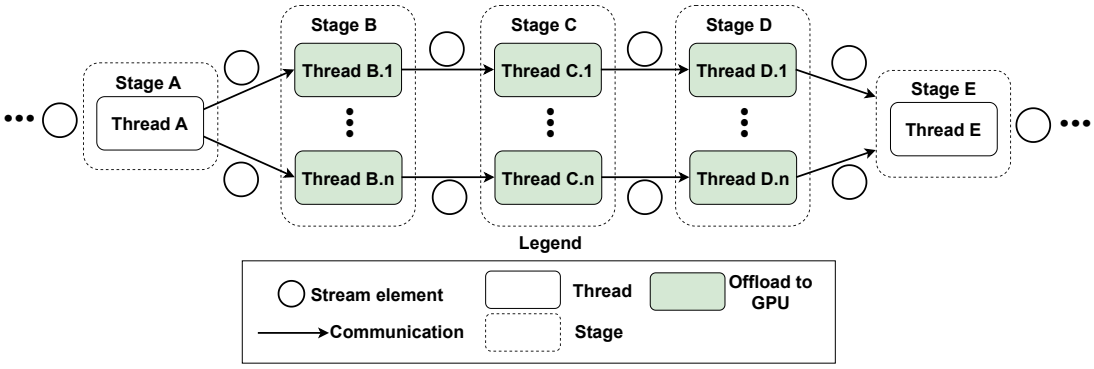

Figure 4.5 – AnimalRescue Flowchart. Borrowed from [23]

LaneDetection (LD) is a computer vision task that involves identifying the boundaries of driving lanes in a video or image of a road scene. The goal is to accurately locate and track the lane markings in real-time, even in challenging conditions such as poor lighting, glare, or complex road layouts. The algorithms typically use a combination of computer vision techniques, such as edge detection, color filtering, and Hough transforms, to identify and track the lane markings in a road scene. This application is useful mainly for autonomous vehicles since they often have an integrated camera that captures several frames per second. A robust LD application can be very complex. It should cover variables such as illumination, appearance, and age of a lane marking [38]. The implementation presented by [52, 53] is a simplified version and does not consider such variables. The code for this application is divided into three stages. The first stage reads an element from the input. The second stage processes an element by applying a Gaussian and a Sobel filter, and the third stage writes an element in the output.

Mandelbrot (MB) is an algorithm that generates fractal images. It is defined by the quadratic polynomial $z = z^2 + c$, where c is a constant and z receives the initial value. Although MB is usually approached as data parallelism when targeting GPUs, Rockenbach [53] adapted it as a stream processing application. In the modified version, each row of the generated image is considered a stream element. The first stage of this application reads an element. The second stage processes the fractal. The third stage writes an element in the output.

The Raytracing (RT) algorithm generates a sequence of frames depicting a CGI scene composed of a pre-defined number of animated spheres with a resolution inputted by the user. The techniques presented in this algorithm are used mainly in applications such as 3D animations and 3D games. Much like the above applications, this is also divided

into three main stages. The first stage is the reading of an element. In the second stage, we process a frame. The third stage sends the frames to the output.

All three applications, LD, MB, and RT, were provided in Rockenbach's master's thesis [52] as a way to evaluate stream parallelism with GSParLib. Because of this, they share many similarities in their flowcharts of execution, being divided into three primary stages as shown in Figure 4.6, focusing especially on leveraging the GPU for the processing required in the second stage of each application.

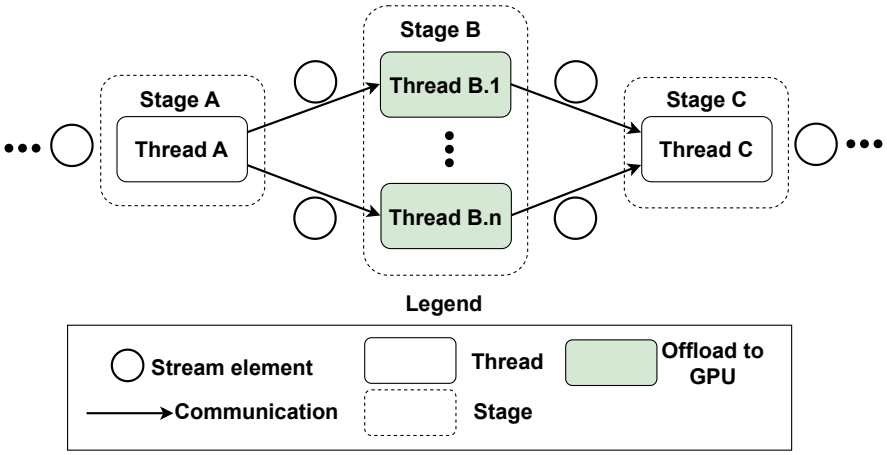

Figure 4.6 – LD, MB, and RT Flowchart. Borrowed from [23]

4.3 Fine-tuning of Scheduling Algorithms with Pthreads and GSParLib

As a preliminary step to the implementations on SPar, we ported versions of the previously described applications utilizing POSIX Threads, since we already have familiarity with the library and had already made a simple stream application that could be easily modified into the described applications, testing rough drafts of our approaches into them, and analyzed the different performances achieved using the policies; in this section, we will only talk about the last drafts and the results obtained when running them into an environment containing multiple GPUs. As an execution environment, we chose a machine running the Ubuntu 20.04.6 LTS operating system, equipped with Intel Xeon E5-2620 (6 cores and 12 threads) and 64 GB of RAM and two GPUs, a NVIDIA Titan X (Pascal) and a Quadro K2200. This machine is the same utilized in Rockenbach's [52] and Gabriell's [23] works, and as such, will work as a suitable comparison to test if our multi-GPU policies are satisfactory. The policies were tested with configurations of 2, 4, 6, 8, 10, and 12 threads in their processing stages. Besides each execution, we also did a solo execution of each application using each GPU with the configurations tested. These dataset used to test each configuration were as follows:

- **AnimalRescue** 2048 drones, 6144 coordinates per drone, 1536 military units per drone, a 2048x2048 land size.
- **LaneDetection** A 1280x720 resolution video with 609 frames.
- **Mandelbrot** A 1500x1500 matrix and 50000 iterations.
- **RayTracing** 500 images of 3840x2160 resolution.

All the graphs follow the same template; the values on the X-axis represent the number of threads that are spawned to the processing stage of the applications; we chose to keep them in pairs, having at least one thread for each GPU present in the environment. The y-axis presents the GPU usage or throughput (elements per second) of the configuration tested depending on the viewed graph. For the graph of GPU usage, we split the multi-GPU execution into two thinner bars, each representing a GPU; they share the same color but have different hatches for each bar. We used the same color to represent the multi-GPU policy on the graphs discussing the throughput of each configuration.

4.3.1 AnimalRescue

The AnimalRescue application was executed using a default workload denominated *WORKLOAD_C*. This workload is comprised of 2048 drones with 6144 coordinates each on a 2048x2048 land; in this configuration, the application must find a suitable point for 1536 animals on each drone.

The AnimalRescue executions, shown in Figure 4.8, display that each policy has very similar usage across the tested configurations, with the *Static* policy showing a more even usage of both GPUs in the environment, both *Saturate* policies did not offload any work to the other GPU, meaning that there were no changes in usage to be detected, so they are not shown in the graph, both *Queue* and *WorkStealing* share similar usages since *WorkStealing* was built on top of the *Queue* work division core policy.

Figure 4.7 displays the throughput obtained by the executions, much like the usage results, the policies share very similar results in the tested configurations, with the *Static* policy having a higher throughput since the data is evenly split between GPUs, *Queue* and *WorkStealing* sharing the same results with some slight variation, and the *Saturate* having somewhat lackluster results since the application was not able to reach the limits of usage to require the offloading of work to the other GPU in the environment. Fortunately, only this application could not somewhat utilize the *Saturate* policy.

The policies independent of the configuration also share very similar runtime, mainly the *Static*, *Queue*, and *WorkStealing* policies. These last two have a slightly higher execution time than the one obtained by the Titan X only execution and much lower than

the Quadro K2200 only execution, while the *Static* policy displayed a similar runtime to the solo Titan X execution. The *Saturate* policy does not follow this trend, with the Weak variation having a higher runtime than its solo counterpart.

In terms of comparison between the policies, for the AnimalRescue application, the *Static* policy displayed a better edge over the other tested policies, having a more evenly distributed usage of the GPUs while also providing a higher throughput of elements during the executions.

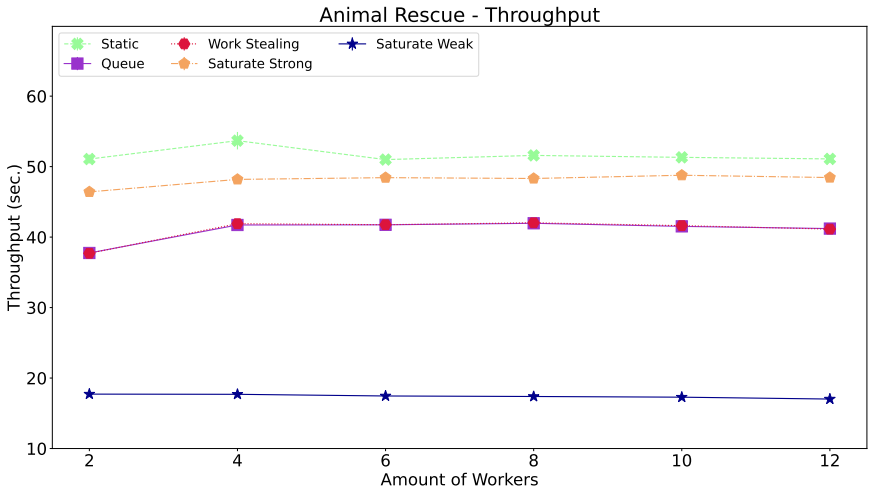

Figure 4.7 – AnimalRescue - Throughput

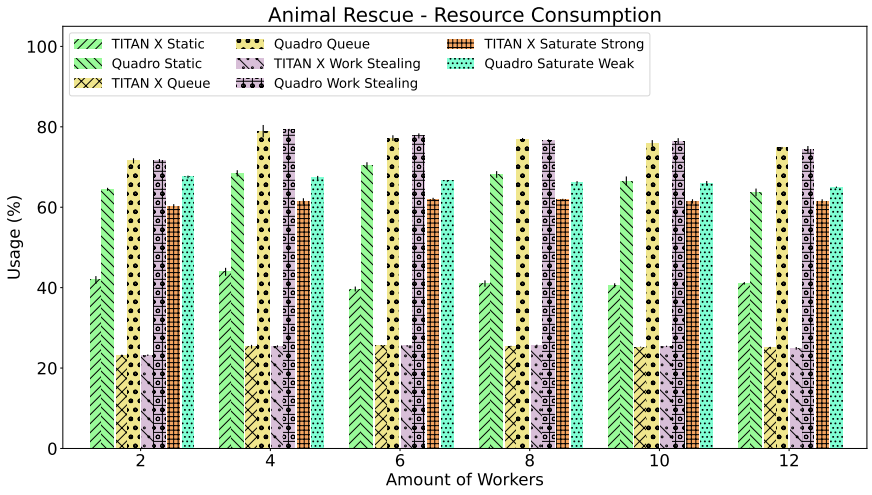

Figure 4.8 – AnimalRescue - Resource Usage

4.3.2 LaneDetection

The execution of the LaneDetection application was done by utilizing a dataset comprised of 609 frames of a 1280x720 resolution video.

The LaneDetection results, shown in Figures 4.10 and 4.9, show that the *Static*, *Queue* and *WorkStealing* policies share similar results in both usage of GPUs and how much throughput each of them can produce. The *Saturate* policy, on the other side, the *Saturate* policy only displayed an insignificant usage of the Titan X on the run seeking to saturate the Quadro K2200, so much that it is almost unable to be seen in Figure 4.10.

Much like the AnimalRescue executions, the policies share a similar runtime between themselves, with the exception of the saturate weak, which achieved a runtime of more than double the other policies.

Leaving this aside, the *Static*, *Queue*, and *WorkStealing* policies also seem to be the wiser choice when running the LaneDetection application, all three sharing the same basic GPU usage and returning comparable throughput between themselves. But we cannot ignore that the *Saturate* in its Strong variation shows a more scalable throughput, obtaining a higher result with the addition of more workers to the configurations.

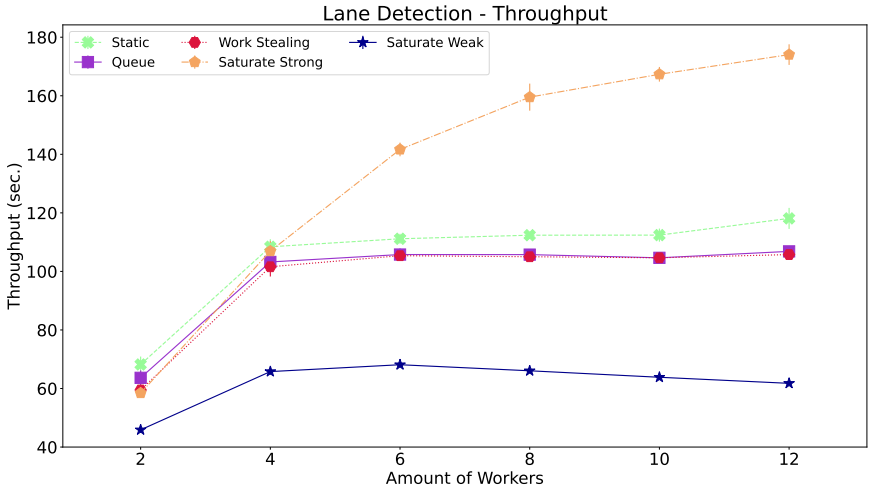

Figure 4.9 – LaneDetection - Throughput

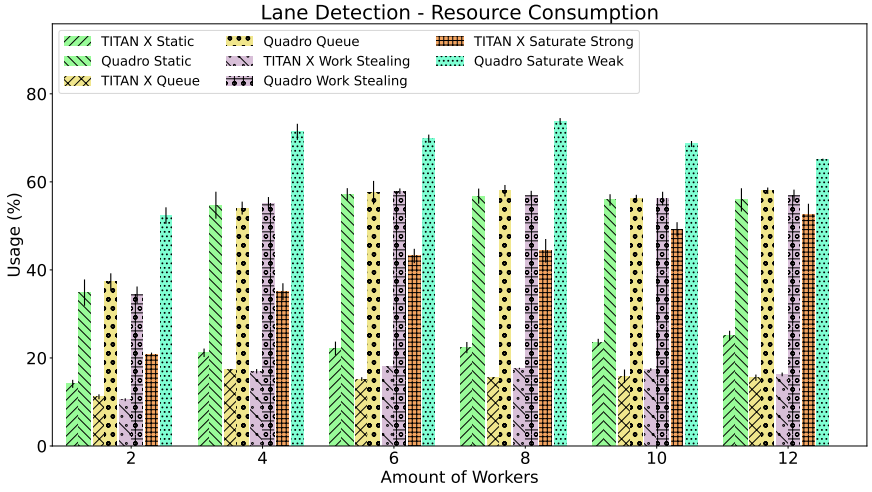

Figure 4.10 – LaneDetection - Resource Usage

4.3.3 Mandelbrot

For the Mandelbrot execution, since there is no recommended workload to test the application, we choose to generate a 1500x1500 fractal using 50000 iterations. The reason behind such high amount of iterations is that MB generates extremely small load for the GPU, and one of the ways to remedy this is by using a high number of iterations.

Figures 4.12 and 4.11 show the execution of the Mandelbrot application with the different policies tested. Much similar to the LaneDetection executions, the *Static*, *Queue* and *WorkStealing* policies shared a similar usage of resources in the different configurations. Both of the *Saturate* policies seem to work perfectly with this application, offloading work to the other GPU when fulfilling the required usage thresholds or when memory is lacking to process a new input.

Like in the AnimalRescue application, the *Static*, *Queue* and *WorkStealing* policies on the MB application share a similar execution time between themselves, with them having a slightly higher execution time than when running the application using only the Titan X, but at the same time displaying a similar time to the execution using only the Quadro K2200.

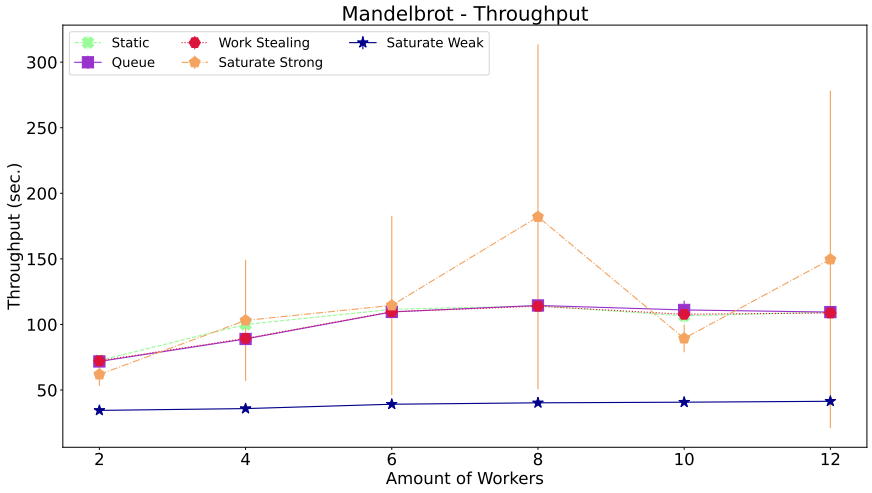

Figure 4.11 – Mandelbrot - Throughput

4.3.4 Raytracing

The execution of the Raytracing application consisted of generating 500 images with a resolution of 3840x2160 pixels, which is 4K UHD. We chose this configuration for its heavy resource usage, especially for the solo execution using the Quadro K2200.

The Raytracing execution, Figures 4.14 and 4.13, much like the LD and MB executions, shows the policies *Static*, *Queue*, and *WorkStealing* sharing a similar GPU usage

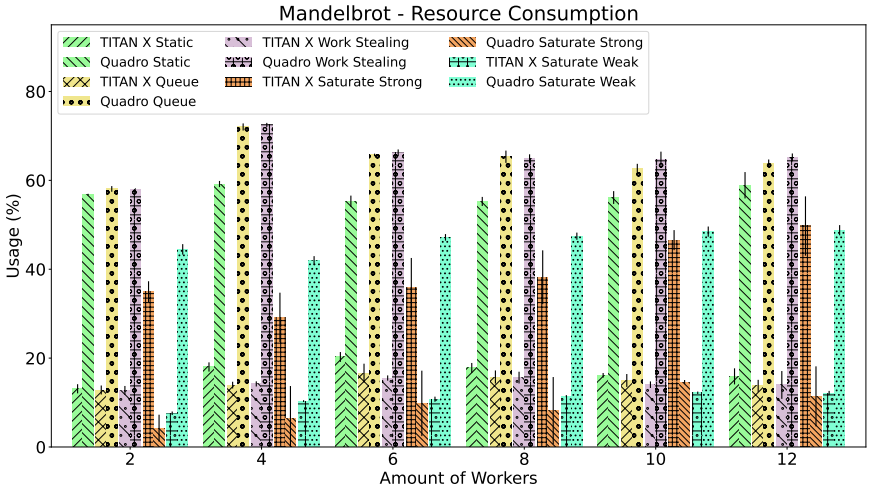

Figure 4.12 – Mandelbrot - Resource Usage

and throughput between themselves, different from the other application the RT was not able to scale in terms of throughput when increasing the number of workers, this is mainly because of the fact that the computations done in the application are executed solely in the GPU, while the other applications have a mix of CPU and GPU computations.

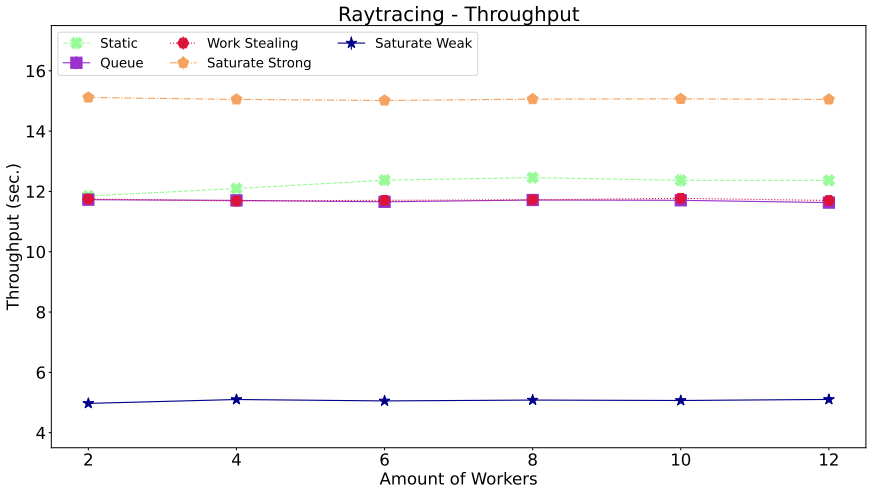

Figure 4.13 – Raytracing - Throughput

The results obtained show that the *Saturate* policy, using its Strong variation, seems to be the most beneficial to the Raytracing execution on multi-GPU, having a similar throughput compared to the other policies but using a lower percentage of the GPU to realize the processing of the elements.

4.4 Final Remarks

This chapter presented the chosen policies of multi-GPU scheduling that were planned to be implemented into SPar's GPU code generation, as well as the applications

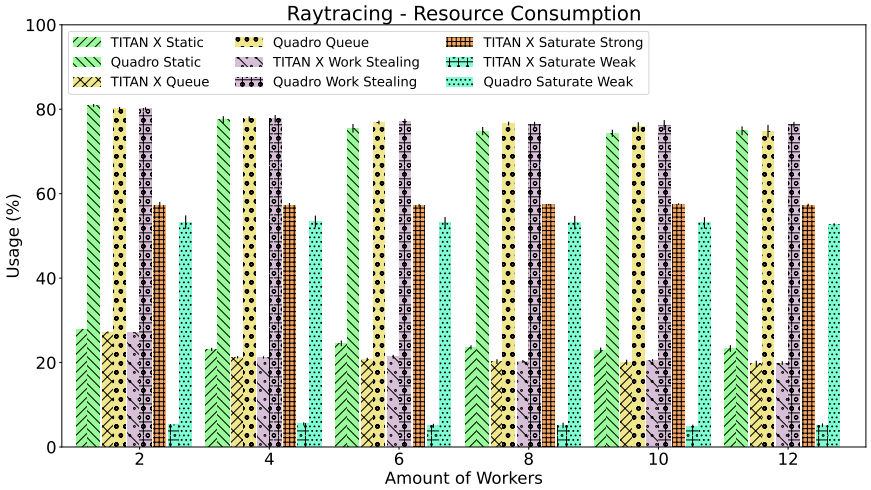

Figure 4.14 – Raytracing - Resource Usage

that were utilized to test if such policies were beneficial when compared against single-GPU executions. Table 4.1 shows that when comparing the policies, in terms of throughput, the *Saturate*, in its Strong variation, shows the most promise, especially in more processing-intensive applications, with the *Static* being a closer contender. The utilization of multiple queues as utilized on the *Queue* approach also does not seem to impact the performance of the applications, and an approach such as *WorkStealing* does not seem to bring any increase in results, and as such, we do not see any usefulness of applying such approach to the usage of multi-GPU. It is difficult to, in terms of resource usage, say which approach is the most promising since *Saturate* seeks to focus work on a single GPU; comparing its resource consumption to other approaches that intend to balance the usage of multiple GPUs seems unfit and unfair with the approach, and since at its core *Queue* and *WorkStealing* use *Static*'s scheduling policy, their results are very similar.

Table 4.1 – Best Results

Benchmark	**Throughput**	**Policy**	**Amount of Workers**
AR	53.68	Static	2
LD	174.05	Saturate Strong	12
MB	182.07	Saturate Strong	8
RT	15.07	Saturate Strong	10

While the *Queue* approach was somewhat based on FastFlow's, its feasibility of implementation onto SPar would require significant restructuring of the code generation as well as changes to FastFlow's load-balancing implementations, which would not be feasible inside the scope of this work, as such we focused on adding the *Static* and *Saturate* scheduling approaches to SPar's code generation, with the *Saturate* appearing to be the most beneficial for the SPar code generation for multi-GPU environments.

5. HIGH-LEVEL MULTI-GPU SUPPORT

This chapter briefly presents the current transformation rules of SPar for multi-GPU code generation, the implementation of the policies into SPar's code generation and the changes done to accommodate the generation of codes needed by the policies, section 5.1.1, as well as executions utilizing the implemented policies in SPar using the previously discussed stream processing applications, section 5.2.

5.1 SPar GPU transformation rules

This section describes the transformation rules and the code generation mechanisms on SPar compiler targeting CPUs and GPUs, and is based on the descriptions provided by [52, 55] about its implementation of GPU code generation on SPar. Figure 5.1 was taken from [55]. It provides an overview of SPar's approach to parallel code generation. The transformation rules can be divided into three steps: parse and extract information from annotated source code, match and apply transformation rules, and perform the actual code generation. The initial step gathers all the information needed for generating code for both the CPU and GPU. The second and third steps have specific implementations that cater to each architecture. The GPU is utilized solely for exploiting data parallelism, whereas the CPU can take advantage of both stream and data parallelism. Therefore, to support a combination of stream and data parallelism using both the CPU and GPU, the compiler first processes these three steps for the GPU and then repeats the second and third steps for the CPU.

The first step, called "Parsing", is divided into two separate tasks: (1) C++ syntax parsing, where the SPar compiler verifies the correctness of the semantics of the annotated C++ code it received as input and generates the AST from the code; and (2) extracting information from the code, where it performs static code analyses on the AST to extract all the necessary information to generate the parallel code. Since GPUs have dedicated memory, SPar must perform data copies between the host and GPU memories whenever necessary to keep the data updated.

The SPar compiler analyzes the Abstract Syntax Tree (AST) to extract essential information for GPU parallelism by considering the following aspects: (1) Data: identifying which data and their sizes are necessary for GPU computations; (2) Source Code: determining which portions of the code will execute as GPU kernels; and (3) Dependencies: identifying the data structures and methods that the GPU kernels utilize during computation. Conversely, achieving CPU parallelism does not require such detailed information since the parallelism regions share the same memory space. To extract information from the AST, the SPar compiler follows these steps:

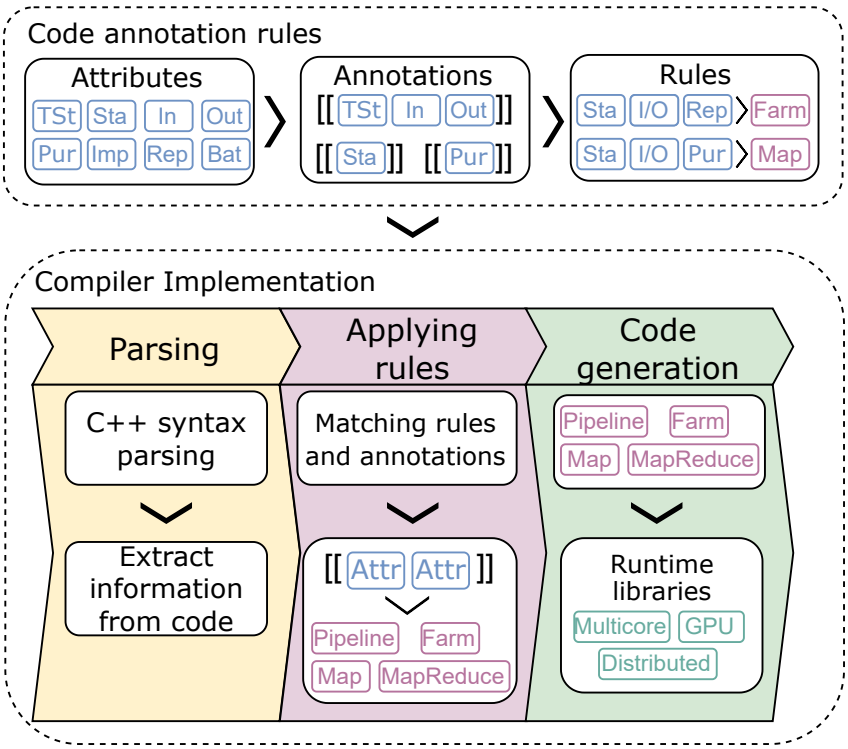

Figure 5.1 – Methodology for Parallel Code Generation.

1. Reference List Creation: It creates a list of references with pointers to the code blocks annotated with an SPar identifier attribute, such as ToStream, Stage, Pure, and Impure, along with their auxiliary attributes. These lists are organized in a tree structure, indicating the hierarchical relationships between the attributes.

2. Information Extraction: The compiler extracts information from the annotated code blocks necessary for code generation for both CPUs and GPUs. For example, upon encountering a block of code containing a Pure annotation, the compiler adds a reference to this code block into an internal stack of Pure annotated code blocks.

3. Recursive List Generation: It recursively generates a list of all non-standard data types and methods referenced within each annotated code block. While generating CPU parallel code requires only the specific code block itself, generating GPU code necessitates mapping all external definitions used by that code block, including data structure definitions and functions, as these must be provided alongside the GPU kernel at compile time.

After extracting the relevant information from the code, the SPar compiler moves on to the second step of its implementation, referred to as "Applying Rules," as illustrated in Figure 5.1. In this step, the SPar compiler iterates through the tree structure of the identifier attributes collected during the information extraction phase and attempts to match them with predefined transformation rules. When a transformation rule is satisfied, the SPar-annotated code is replaced with a code block that calls a Map or MapReduce

parallel pattern. A parallel pattern is essentially a function that takes the code block, its dependencies, along with inputs and outputs (all of which were identified in the previous step), as parameters. This phase is crucial in determining which parallel pattern should be generated.

Following this, the definition of the chosen parallel pattern is employed to execute the actual source-to-source code transformation, converting the parallel pattern into routines that are specifically designed for the underlying hardware. It is important to note that CPUs and GPUs require entirely different routines during this code generation process.

Figure 5.2 outlines the logic used by the SPar compiler to match the transformation rules. For each ‘ToStream‘ annotation, the compiler attempts to transform it into a purely data-parallel form using either Map or MapReduce. A match is confirmed when the code block annotated with ‘ToStream‘ is also marked with either Pure or Impure and does not contain any code blocks annotated with Stage. This verification is indicated as “Is pure data parallelism?” in the flow diagram of Figure 5.2.

If this check yields a positive result, the entire code block annotated with ‘ToStream‘ is replaced by a node that calls a parallel pattern for data parallelism. If the code block has an Impure attribute, the compiler generates a MapReduce pattern; otherwise, it generates a Map pattern. This completes the compilation step.

If pure data parallelism is not found, the compiler proceeds to examine the list of Stage attributes within the ‘ToStream‘ region. For each Stage, it checks if the Pure attribute is present in the Stage code block. This step is noted as “Is Stage Pure?”. If the Pure attribute is present, the compiler replaces the entire Stage code block with one that invokes a Map or MapReduce pattern. If the multi-GPU flag is set, the Stage receives additional transformations targeting the usage of multiple GPUs depending on which policy was chosen, if the flag is not set, no additional transformations are required and the compilation continues as normal.

If the Pure attribute is absent, the compiler then checks for any Pure attributes that serve as identifier attributes within the Stage code block. If a match occurs, the compiler transforms the Pure code block into one that calls a Map or MapReduce pattern, depending on whether the Impure attribute is present. After processing all Stage, Pure, and Impure attributes, this step concludes its execution.

After reviewing all the transformation rules for data parallelism, the compiler resumes its original code generation algorithm for stream parallelism in SPar. In this phase, the compiler applies the original transformation rules using the remaining SPar annotations in the code to generate stream parallelism, specifically Farm and Pipeline. This approach allows for the combination of stream and data parallelism since a Pipeline stage can include computations executed on the GPU.

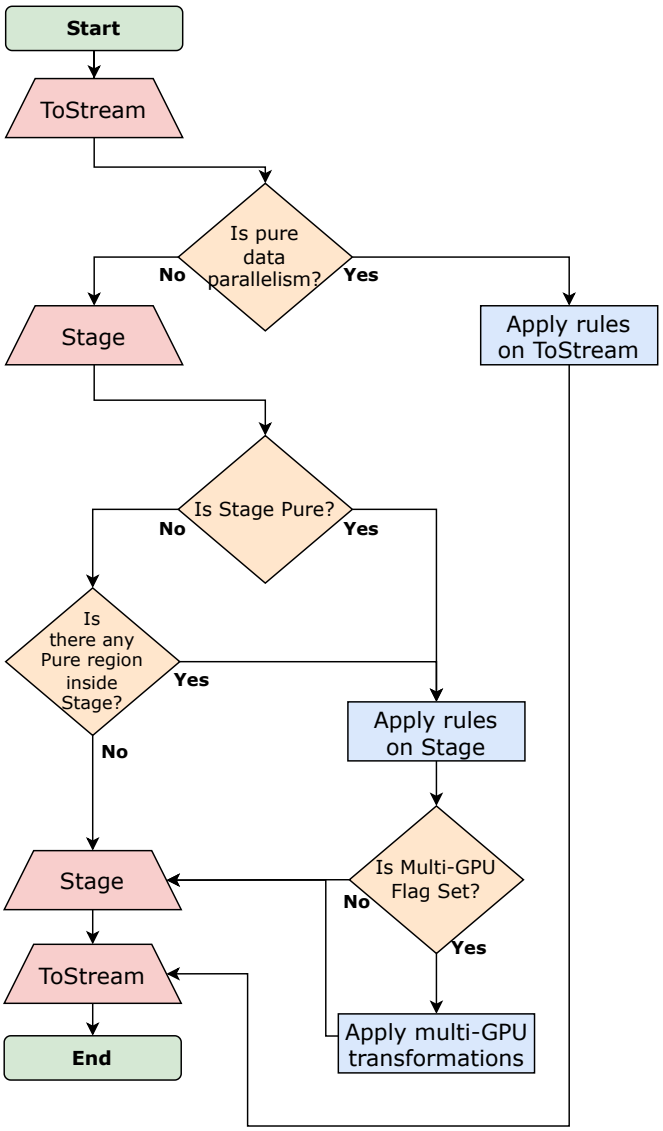

Figure 5.2 – Flow of the transformation rules of SPar. Adapted from [52]

Following the extraction of information from the source code in the first step and the matching of transformation rules in the second step, the compiler advances to the third and final step, as illustrated in Figure 5.1: “Code Generation.” The main objective of this step is to replace the parallel pattern calls produced in the previous step with low-level code optimized for the target hardware, whether it is the CPU or GPU.

The system uses information from the for loop statement to generate Map and MapReduce patterns. This includes details such as the iterator variable name, the starting and ending expressions, and the loop body. When targeting the GPU, it creates a parallel pattern optimized for GPU execution using GSParLib, assigning a GPU thread to each iteration. For CPU targeting, it generates a parallel pattern optimized for CPU using FastFlow, statically assigning a group of iterations to each thread. Users can select the number of threads by setting an environment variable; if not set, the default is the number of physical CPU cores.

For MapReduce code generation, the code block annotated with the Impure attribute is analyzed to extract the application variables that will be reduced. When targeting the GPU, definitions of data types and functions referenced in the for loop body are incorporated into the GPU kernel code for compilation. However, when targeting the CPU, this information is not included. The compiler generates the Map and MapReduce parallel patterns and stores them as global objects. It replaces the pure code blocks with calls to these patterns and inserts all necessary definitions at the beginning of the source code, including additional structs and headers for the underlying runtime backends to effectively leverage CPU and GPU parallelism.

After generating the data parallel patterns for the GPU, the compiler begins the CPU code generation process. This process starts with generating parallel patterns focused on data parallelism (such as Map and MapReduce) and then proceeds to generate patterns for stream parallelism (including Pipeline and Farm). This approach has two main benefits:

1. If a ToStream region can be represented as either a data or a stream parallel pattern, it is generated as a data parallelism pattern. This is because data parallelism typically involves less coordination overhead and enables greater speedups compared to stream parallelism.

2. When a data parallel pattern is generated for a Stage or Pure region, the transformations focused on streaming parallelism effectively "wrap" this pattern into a Pipeline stage or Farm worker during the generation of these patterns. This allows for a seamless combination of stream and data parallelism, utilizing both CPU and GPU parallelism.

By generating GSParLib code, the SPar compiler takes advantage of its thread safety, synchronization mechanisms, and automatic memory transfers. As a result, only minor adjustments to the original compiler algorithm for stream parallelism are needed to integrate the new GPU code generation process effectively. One such adjustment involves the compilation of the GPU kernel. Since GSParLib uses runtime compilation, which can significantly degrade performance if executed repeatedly, the compiler moves the GPU kernel compilation step to occur before the stream region (outside the loop). It also clones the GPU pattern objects for each CPU thread to reuse the compiled kernel, avoiding additional overhead costs.

5.1.1 Changes on SPar’s code generation

Since our main focus was adding the generation of code targeting multi-GPU into SPar we focused our efforts on first learning how the annotation transformations are

done inside of SPar. Thankfully, SPar has a clear division of files, with the transformations targeting GPUs having their own separated files, called *gpu_transformation.hpp* and *gpu_transformation_definitions.hpp*; these two files contain everything regarding the transformation of the annotated code into GPU capable code. As demonstrated before, the two best proposed scheduling policies were *Static* and *Saturate*, therefore, they were chosen to generate inside SPar compiler.

Figure 5.3 shows an example, borrowed from [52], of code generated using GPU transformations of SPar, for simplicity we colored the parts added when using the *Static* policy in red. The addition of the policy into SPar's code transformation was very simple since GSParLib already offers a method of setting the desired GPU for each generated kernel, as well as a method that returns the number of GPU present in the environment, together with a method provided by FastFlow that returns the ID of the worker thread, allowed easy construction of the needed code for multi-GPU usage.

While not previously talked when talking about the *Saturate* policy, one of its main needs is a way to retrieve information regarding the usage of each GPU; for this, we tested different ways of getting such data, like querying the $nvidia-smi$ command to extract the needed data or even using the CUDA API, but these methods prove themselves very slow, and as such we choose to use the NVIDIA Management Library, a C-based programmatic interface for monitoring and managing various states within the GPUs; unfortunately, this library only works when running using NVIDIA accelerators which limits the usability of the policy.

Regarding the implementation of the policy into the code generation phase, we choose to implement its main component into the FastFlow's Emitter component since the usage query could become a bottleneck if multiple workers try to get the information regarding GPU usage at the same time, as well as put a strain on the GPUs. Figure 5.4 shows an example of code generated using the *Saturate* policy, for simplicity, we split the generated code into two parts, the code regarding *GPU Usage Retrieval* is appended at the beginning of the generated code, and account to the needed methods for retrieving the usage data from the GPUs, much like the previous figure, all the code relative to the *Saturate* policy is highlighted in red.

As a simple way of allowing the usage of multi-GPU into SPar, we choose to wrap the code generation utilizing a compilation flag, we called this flag *USE_MULTI_GPU*. This flag, when set, generates the *Static* policy, when used on code targeting GPUs, we also use a second flag that only works in conjunction with the first, *SATURATE_GPU*, that when set generates the needed codes for the *Saturate* policy. This means that no new annotation is required to leverage multi-GPU, meaning that any code previously done in SPar that leveraged GPU can be recompiled to use multi-GPU without changing the annotated serial code.

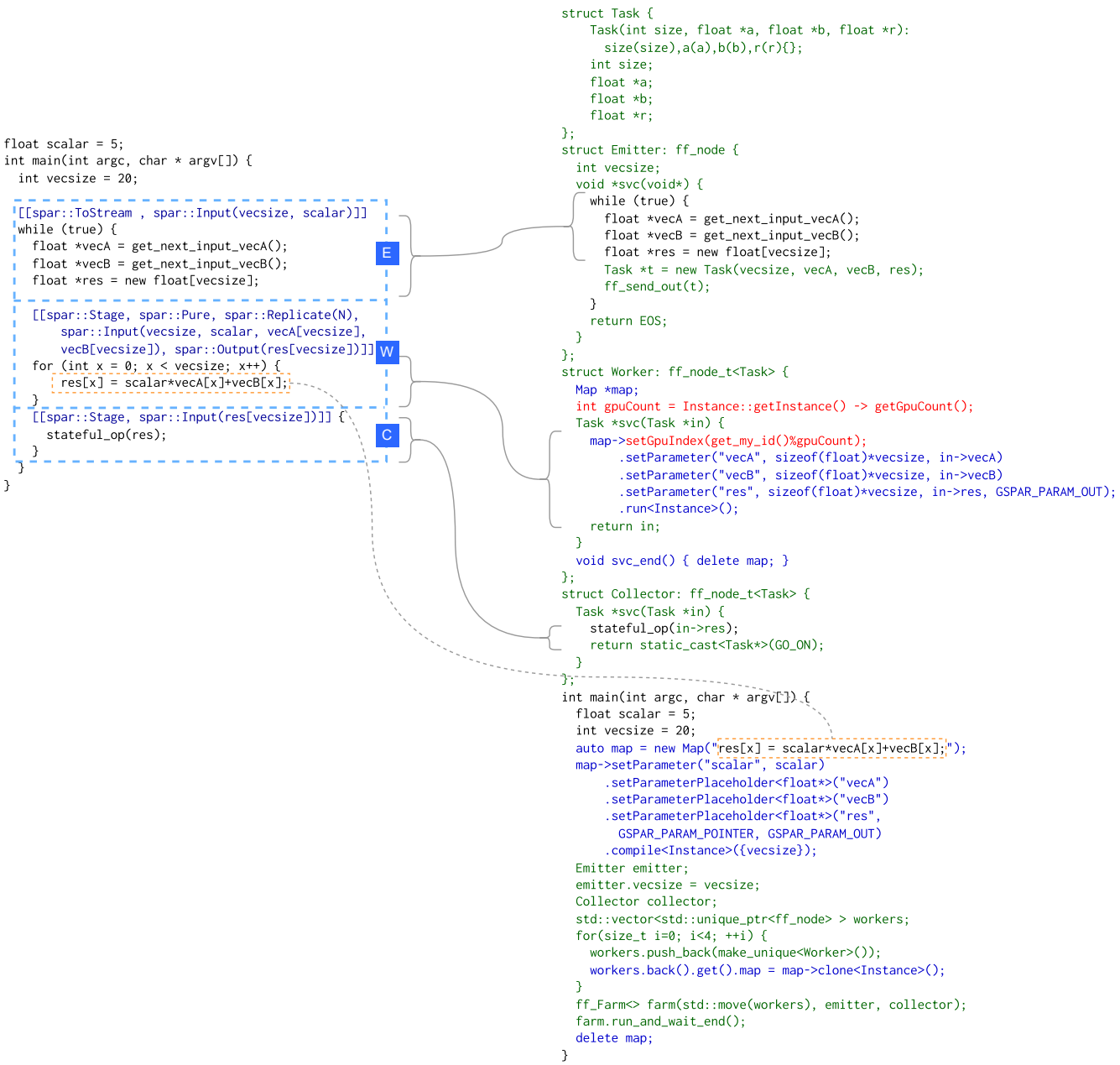

Figure 5.3 – Static code example. Based on [52]

5.2 Results on SPar

We performed experiments using the same methodology and machine described in Section 4.3. We compile each of the applications in SPar utilizing the latest version of FastFlow, version 3.0.0, as well as the flags *−DBLOCKING_MODE* and *−DNO_DEFAULT_MAPPING*, the first enables blocking run-time on FastFlow, and the second tells FastFlow to avoid thread pinning when running the applications, which could impose a performance degradation when executing some of our tests. For all the tests we utilized the code generated by SPar, with the only exception being the single executions, as SPar only generates

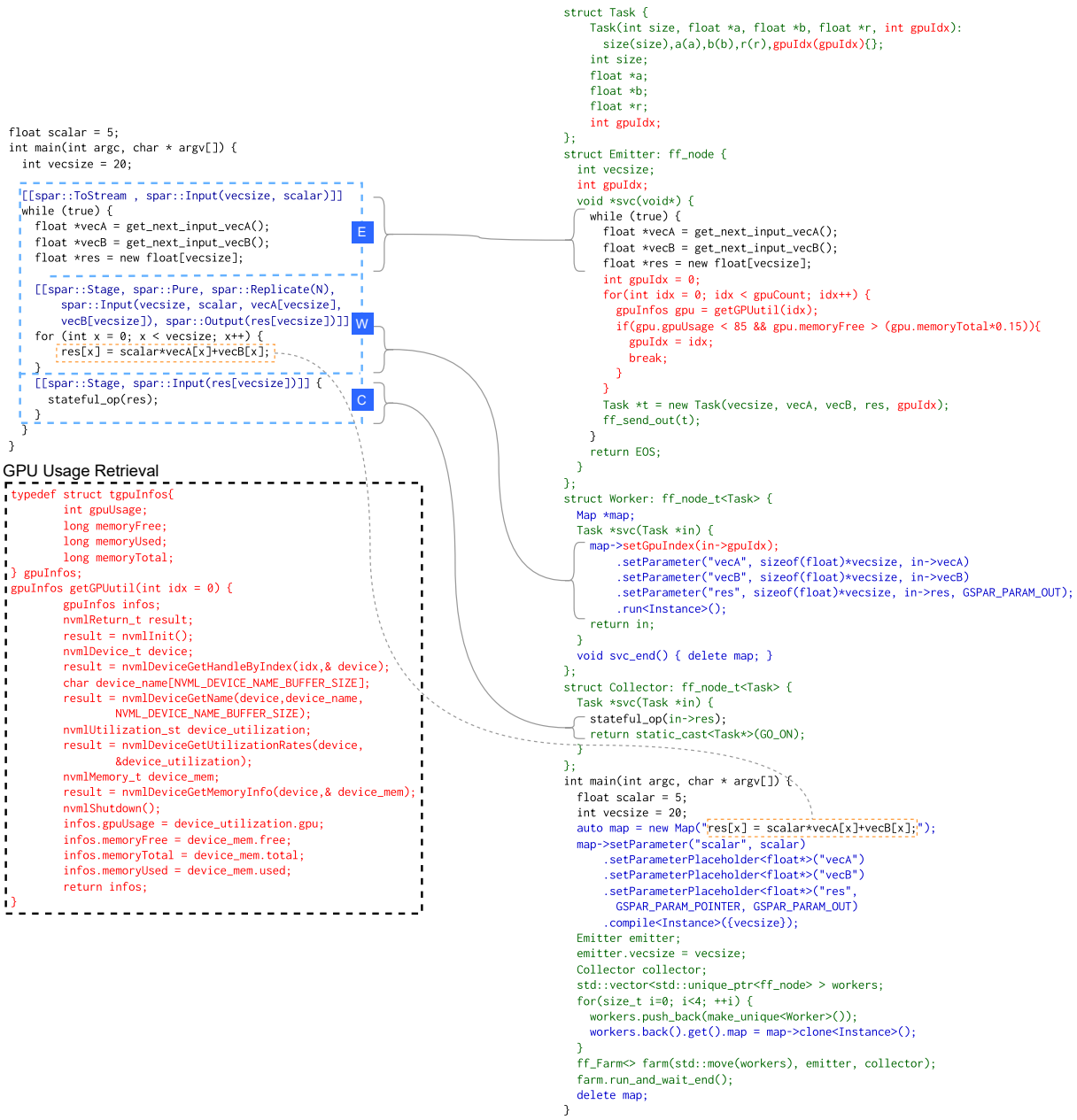

Figure 5.4 – Saturate code example. Based on [52]

code targeting the first GPU present in the environment, we modified the generated code to include the call of GSParLib method that changes which GPU is going to be utilized for processing. We divided the workloads into two classes, B and C. The classes of each benchmark are composed of the following parameters:

- **AR.B.** 2048 drones, 3072 coordinates per drone, 1024 military units per drone, a 2048x2048 land size.
- **AR.C.** 2048 drones, 6144 coordinates per drone, 1536 military units per drone, a 2048x2048 land size.
- **LD.B.** A 640x360 resolution video with 248 frames.

- **LD.C.** A 1280x720 resolution video with 609 frames.
- **MB.B.** A 1500x1500 matrix and 50000 iterations.
- **MB.C.** A 2000x2000 matrix and 100000 iterations.
- **RT.B.** 1000 images of 2560x1440 resolution.
- **RT.C.** 2000 images of 3840x2160 resolution.

Figures 5.5, 5.7, 5.9 and 5.11 illustrate the number of elements processed per second (throughput) in the LD, MB, RT, and AR benchmarks. The x-axis denotes the number of worker threads, corresponding to each stage's replicas (where applicable). For instance, we can replicate stages B, C, and D in the AR benchmark, as well as stage B in the other benchmarks. If we set the worker threads to 2 for the AR benchmark, the application will create two copies of stages B, C, and D, resulting in a total of 6 threads, in addition to one in stage A and another in stage E. Concerning the versions: 1) We tested the *Static* and *Saturate* policies with SPar; 2) We executed each benchmark using only a single GPU with SPar, seeking to compare them against our policies. Similar to our preliminary tests, we also took the usage of each GPU during the executions, seeking to compare how each GPU is used when targeted by our policies, shown in Figures 5.6, 5.8, 5.10 and 5.12.

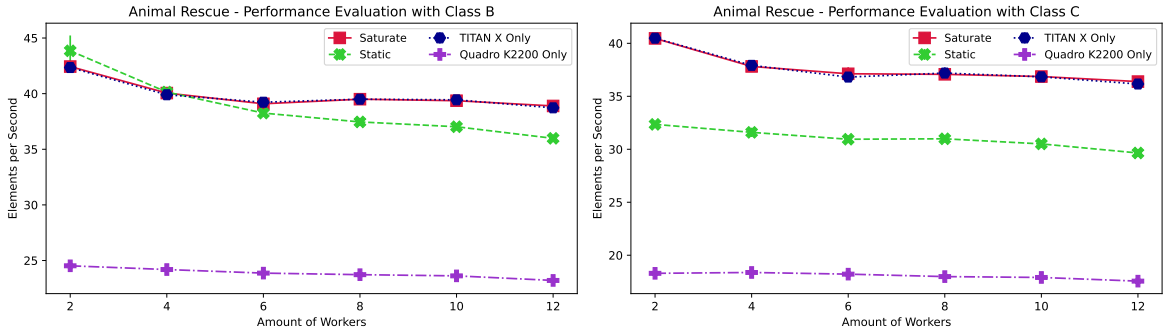

Figure 5.5 – AnimalRescue - Throughput Evaluation

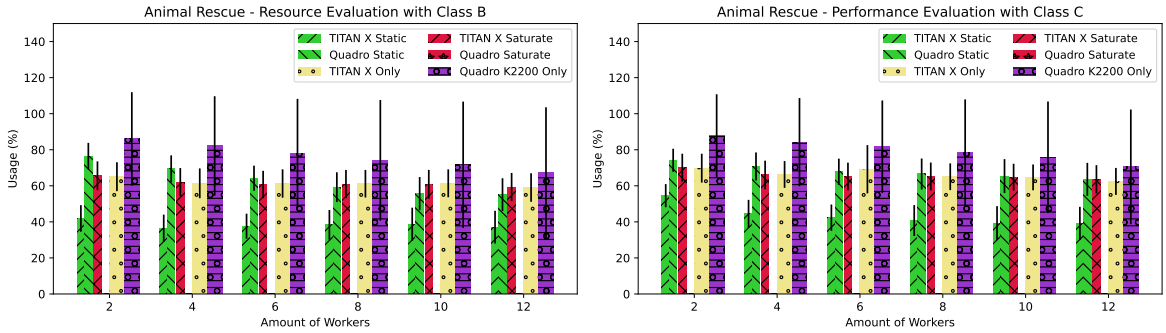

Figure 5.6 – AnimalRescue - Resource Consumption Evaluation

In the AR benchmark (Figures 5.5 and 5.6), the results show that no relevant improvement was achieved by the usage of multi-GPU over a parallel single-GPU version

using the stronger GPU in the environment (*Titan X Only*), with the *Saturate* policy showing similar results in throughput and resource consumption to the single-GPU version using the Titan X in both workloads, while the *Static* show promising results in regards to GPU usage it lacks throughput when using a higher amount of workers which could be explained by its equal division of work between the workers, and in consequence being hindered by the workers that use the Quadro K2200, this is corroborated by the lower throughput also received by the execution using only the Quadro K2200 when using higher amounts of workers. It is imperative to say that; unfortunately, SPar was not able to generate a viable version of the AR benchmark through its annotations, and as such, we chose to implement the benchmark using FastFlow, mimicking the way that SPar generate the compiled code.

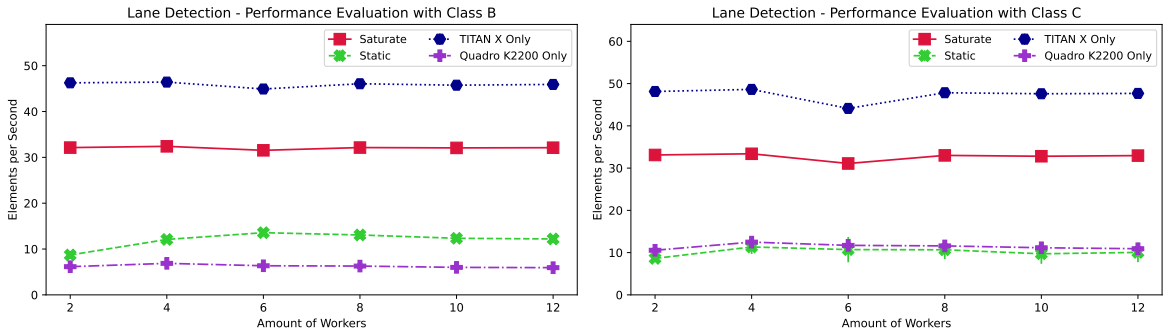

Figure 5.7 – LaneDetection - Throughput Evaluation

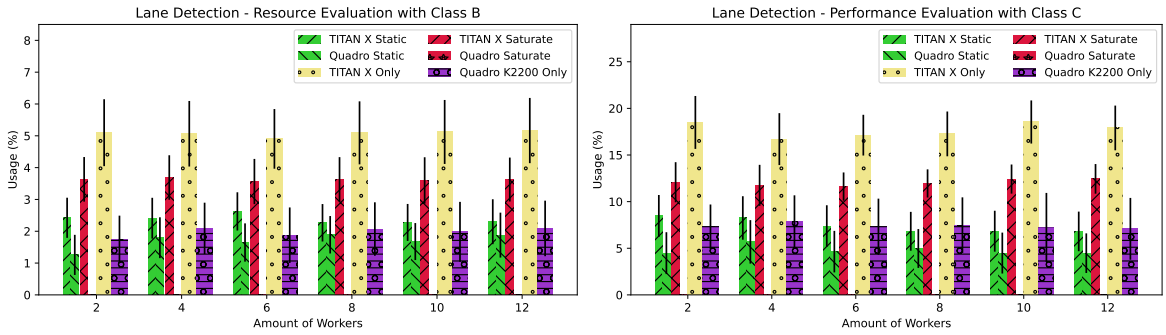

Figure 5.8 – LaneDetection - Resource Consumption Evaluation

The LD benchmark (Figures 5.7 and 5.8), does not show any improvement by the usage of multi-GPU with the tested workloads; this could be caused by its high amount of communication between the CPU and GPU, with the application applying three separate transformations via GPU with more transformations done by the CPU in-between the GPU calls. *Static* results closely mimic the results obtained by using only the Quadro K2200; this signals that the division of work done in this policy is highly hindered when using GPUs with different computational powers since the benchmark does not explore robust strategies or GPU resources the GPU with lower power dictates the throughput. *Saturate* suffers a similar fate, with a lower throughput compared to a Titan X only execution of the benchmark; this is mainly due to the time expended consulting the GPU usage when

sending works; since the computations are very simple, the time used to decide which GPU to use quickly adds up and hinders the results of the executions. This is also reflected in the GPU resource consumption, with workload B having one of the smallest usages of all the applications and workload C also not reaching high usages.

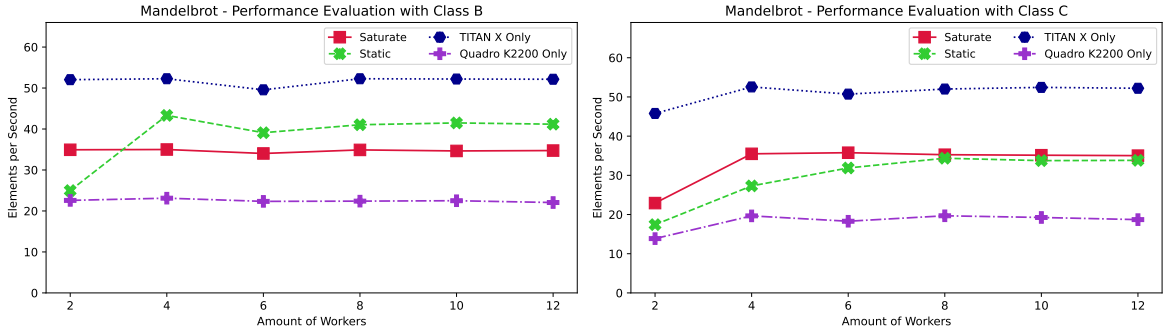

Figure 5.9 – Mandelbrot - Throughput Evaluation

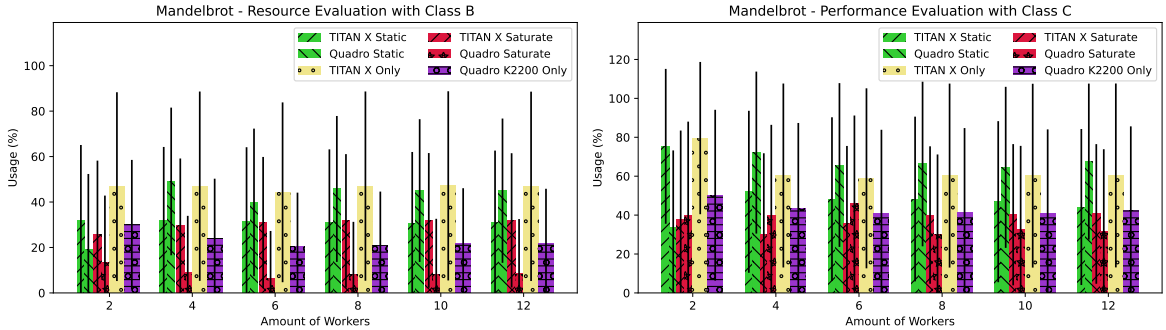

Figure 5.10 – Mandelbrot - Resource Consumption Evaluation

Every limitation of the LD benchmark is evident in the MB benchmark (Figures 5.9 and 5.10). However, MB generates even smaller loads for the GPU than LD because computing a matrix row is less computing-intensive than processing a whole frame. Although both *Static* and *Saturate* show similar throughput in both workloads, they are still lower than just using solely the Titan X. Still, because of the computational simplicity of the benchmark, *Static* was not so hindered by the lack of computational power provided by the Quadro K2200. In a question about resource usage, it shows a higher usage than the LD benchmark executions, although with very large standard deviations; this could be explained by our tests using extremely large workloads to compensate for the lower computational load present in the benchmark. It is essential to note that only the Mandelbrot was able to reach the required threshold of *Saturate* to require the offloading of work to the second GPU, which could also explain its results being a middle-ground between the single GPU executions.

The RT benchmark (Figures 5.11 and 5.12) is similar to the LD and MB benchmark, with RT performing fewer communications between the CPU and the GPU than LD and has larger loads for the GPU than MB. The policies also do not seem to show any improvement

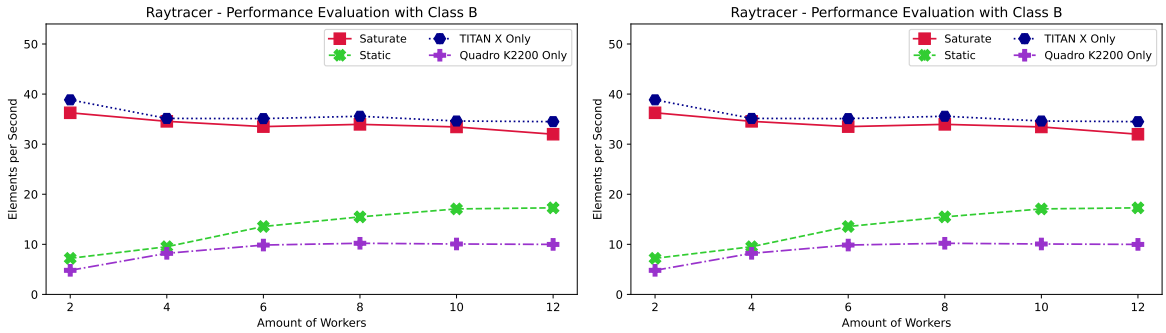

Figure 5.11 – Raytracer - Throughput Evaluation

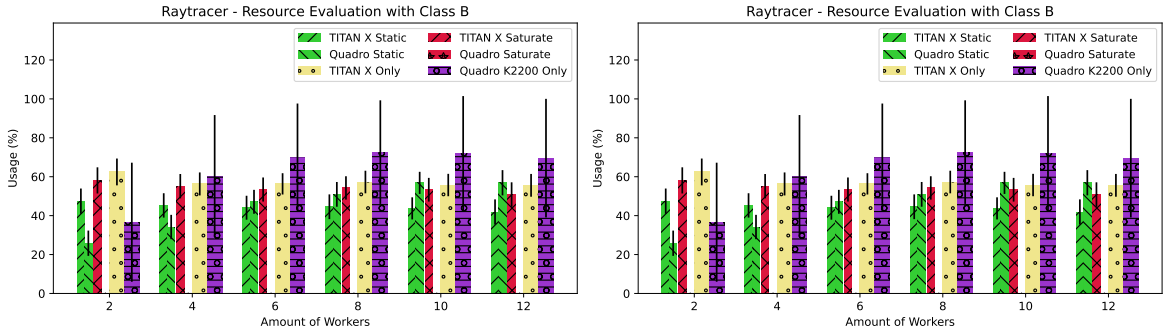

Figure 5.12 – Raytracer - Resource Consumption Evaluation

to the executions, with *Static* being impacted by the lower power of the second GPU and showing slightly higher throughput than only using the Quadro K2200, and *Saturate* having similar results to only using the Titan X.

5.2.1 Tests utilizing on-demand scheduling on SPar

Although we could not implement the *WorkStealing* approach into our policies thanks to limitations imposed by FastFlow, we realized that FastFlow already has a method call that turns the executions from a Round-Robin scheduling into an On-Demand scheduling, in which the workers spawned by FastFlow request works from the Emmiter thread, this approach is very similar to our intent when using the *WorkStealing*, dividing the workload in a more even matter, with the GPU(s) with higher computational power taking a higher share of work, and can be easily set up via SPar by compiling the code utilizing the flag $-$*spar_ondemand*. Figures 5.13 to 5.20 show the throughput and resource consumption of the on-demand scheduling when using multi-GPU with the applications.

The AR benchmark does not achieve any improvement in a general sense when using on-demand scheduling, with the exception of the *Static* that shows a higher throughput than the round-robin scheduling (Figure 5.5) used by default in FastFlow, while all the other executions kept the same throughput. Resource usage was also kept the same when using on-demand, with the exception of the *Saturate* that reached its threshold of offload-

ing with 12 workers, causing the throughput to also drop thanks to the lower capability of the second GPU.

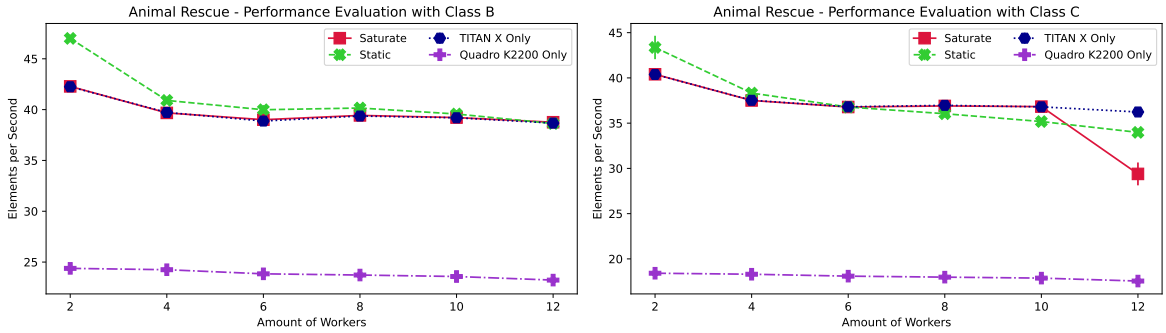

Figure 5.13 – AnimalRescue On-Demand - Throughput Evaluation

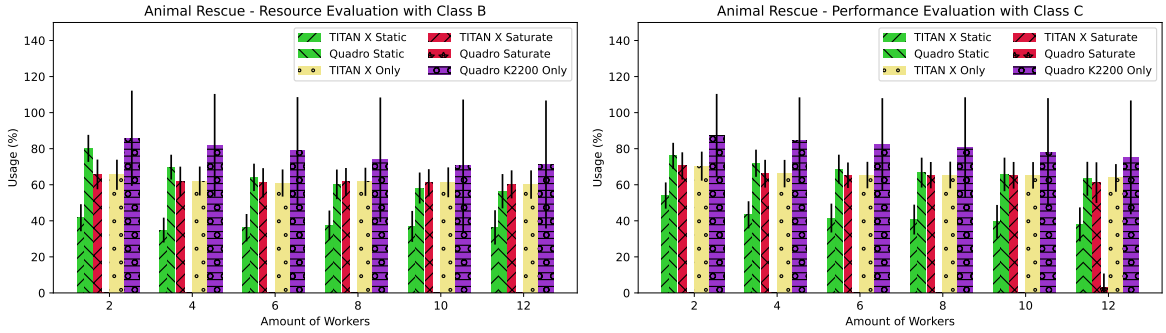

Figure 5.14 – AnimalRescue On-Demand - Resource Consumption Evaluation

The LD benchmark also does not achieve any improvement outside of the higher throughput gained by the *Static* policy when running utilizing the on-demand scheduler, but differently from the other applications, even this is not enough to pass the throughput obtained when running singly using the Titan X, as well as introduce a lot of deviation to the *Static* policy, since it does not take in mind the current consumption of the GPUs when requesting a work.

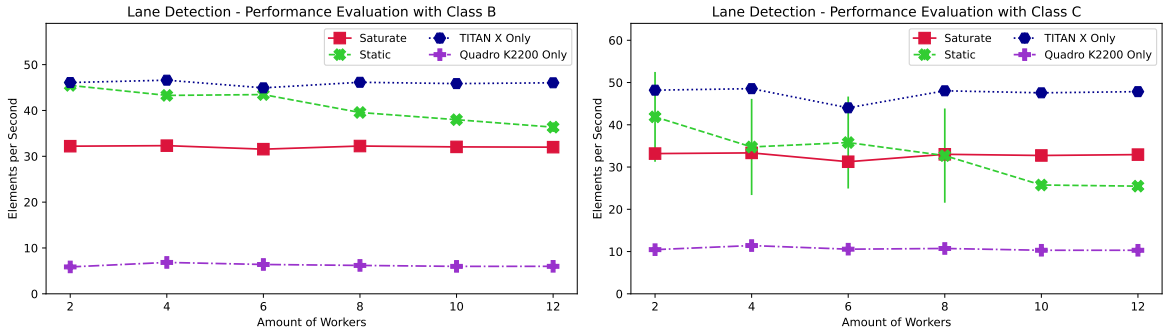

Figure 5.15 – LaneDetection On-Demand - Throughput Evaluation

Just like the other benchmarks, the MB benchmark executions, with the exception of the *Static* policy, do not gain any improvement when using the on-demand scheduling.

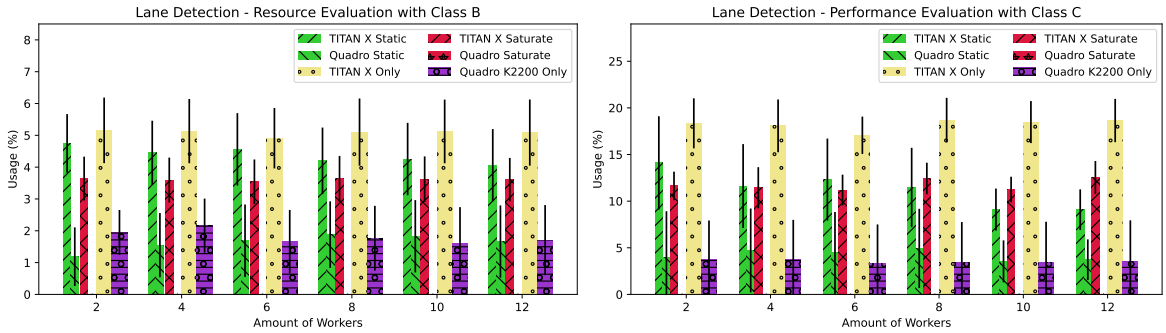

Figure 5.16 – LaneDetection On-Demand - Resource Consumption Evaluation

Resource usage was also very similar to the default round-robin executions (Figure 5.10), although with a much higher standard deviation for the workload B.

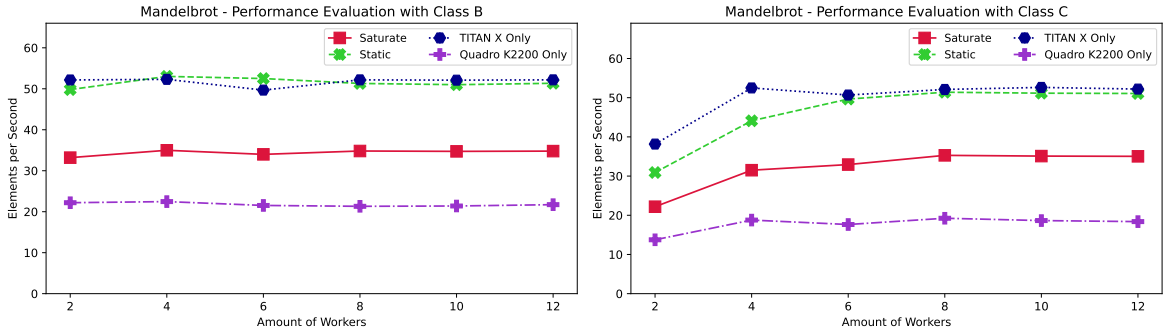

Figure 5.17 – Mandelbrot On-Demand - Throughput Evaluation

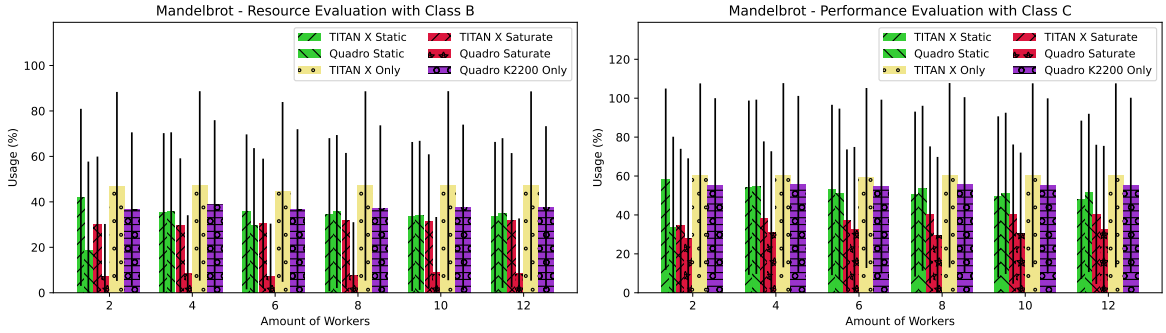

Figure 5.18 – Mandelbrot On-Demand - Resource Consumption Evaluation

The RT benchmark, much like the other benchmarks, only showed an improvement by using the on-demand scheduling in the *Static* policy, achieving the highest throughput in both workloads when comparing the resource usage of the on-demand (Figure 5.20) against the round-robin (Figure 5.12) is possible to note what we expected of this policy, an equal level of utilization of both GPUs.

The tests using on-demand reveal that only the *Static* policy have achievable improvements, since it is mainly based on an equal distribution of GPU per worker, meaning that it is highly influenced by how the workers are distributed between the workers, and the round-robin scheduler utilized by default in FastFlow does an equal division of work

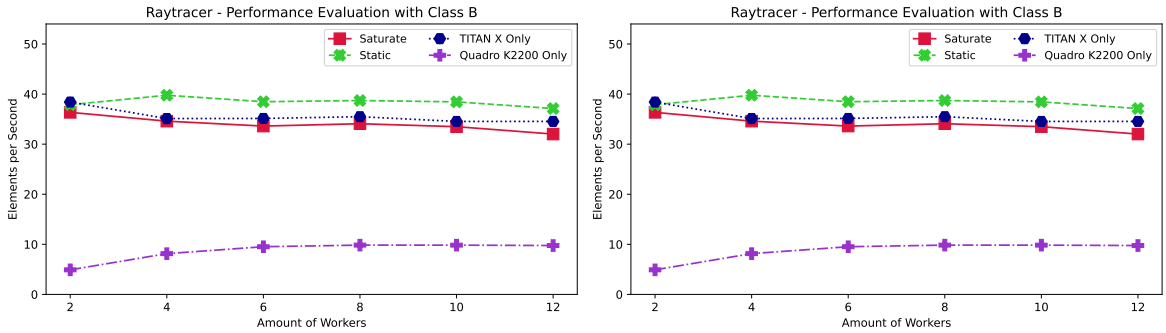

Figure 5.19 – Raytracer On-Demand - Throughput Evaluation

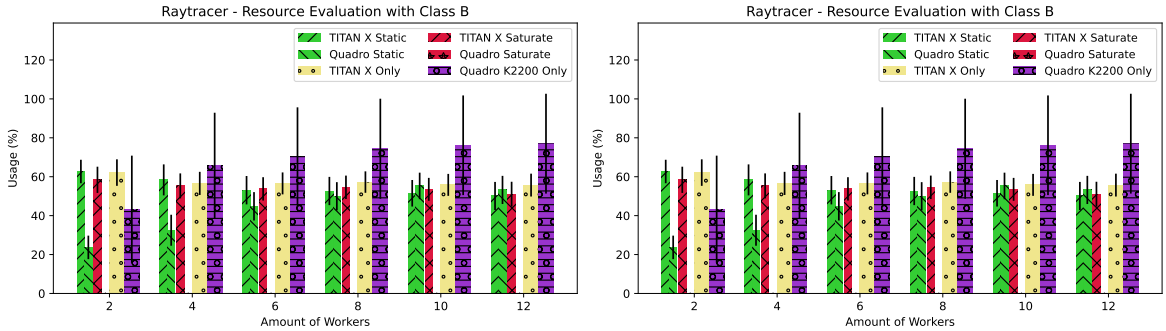

Figure 5.20 – Raytracer On-Demand - Resource Consumption Evaluation

between workers in environments with distinct GPUs the workers assigned with the less powerful GPU becomes a bottleneck in terms of throughput, by utilizing the on-demand scheduler the workers assigned the weaker GPU do not receive tasks while their current task is being performed.

5.2.2 Multi-GPU with batch optimization on SPar

Besides testing the policies in broader workloads than our manual tests, we also tested the usage of multi-GPU with other processing strategies supported via SPar's code annotations, the main being a batch approach to bring the stream parallelism of the benchmarks closer to the data parallelism that is typically used when working with GPUs by grouping a determined amount of work into a single transfer for the GPUs. Unfortunately, only the Mandelbrot benchmark could be executed using the batch approach since the other benchmarks did not meet the required rules for batch optimization, this being that the entire stage must be annotated as *Pure*. The throughput of such execution is shown in Figure 5.21, and its GPU usage is shown in Figure 5.22.

For the tests using batch, we tested using batch size in multiples of 10 inputs. All the tests obtained very similar results, and as such, we chose to show only a single test to avoid repetition when talking about the tests; the chosen execution uses a batch size of 50 inputs. By using batch, both of the single-GPU executions show similar results to each

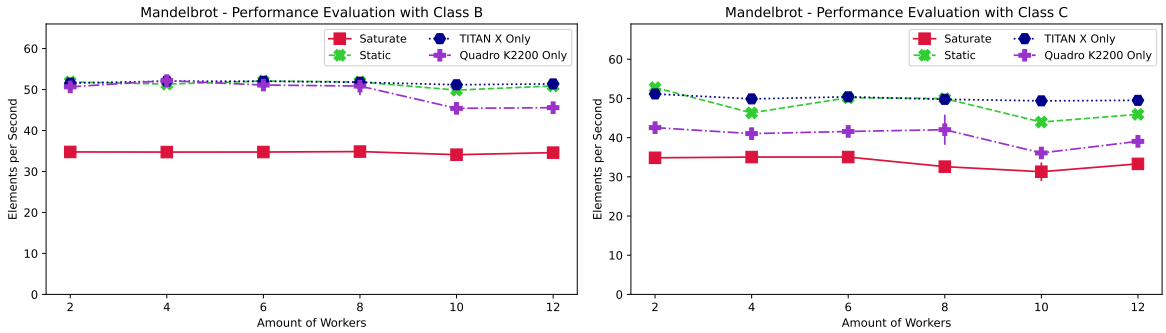

Figure 5.21 – Mandelbrot with Batching - Throughput Evaluation

other, which helps the *Static* policy to obtain higher results than the normal streaming execution done by SPar, *Saturate* still suffers the limitations of needing to choose a GPU to use, which bring its throughput down. Compared to the non-batch execution of MB (Figures 5.9 and 5.10) the resource consumption by using batch is kept low since most execution time is used grouping the required batch, the usage is diluted with periods of no GPU usage, corroborated by the still somewhat high standard deviation present in the execution.

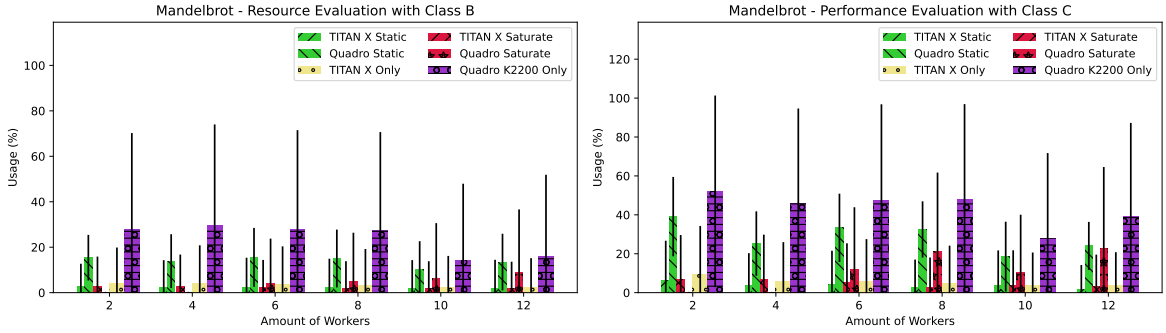

Figure 5.22 – Mandelbrot with Batching - Resource Consumption Evaluation

5.2.3 Batch with on-demand scheduling

We also combined both strategies of batching the tasks into a single transfer to the GPUs and using FastFlow's on-demand worker scheduling, since only Mandelbrot could be executed using the batch approach, our results of combining both strategies are limited to only this application, but it already give us an idea of how this combination could perform in other applications. Figure 5.23 shows the throughput of the execution, while Figure 5.24 shows the GPU usage.

There was not much difference in results by using both batching and an on-demand scheduler, which was somewhat expected since the main drive of using the batch approach is to reduce unnecessary exchanges between CPU and GPU, it also hinders the execution by limiting the amount of data that can be processed in a determined time unit.

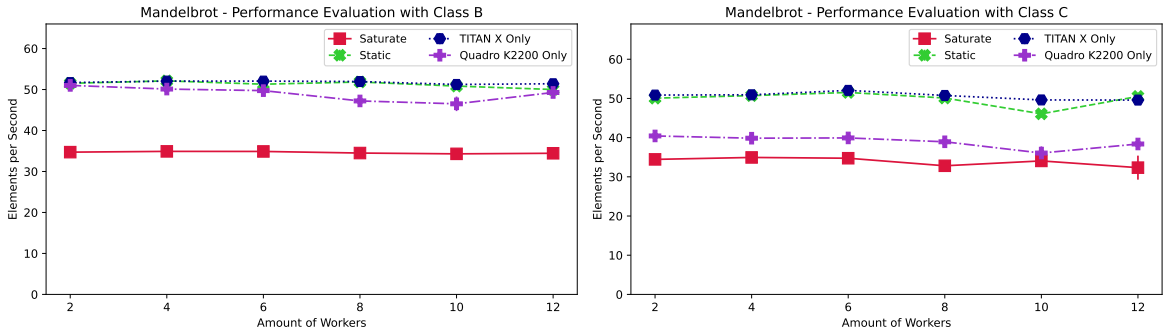

Figure 5.23 – Mandelbrot with Batching and On-Demand - Throughput Evaluation

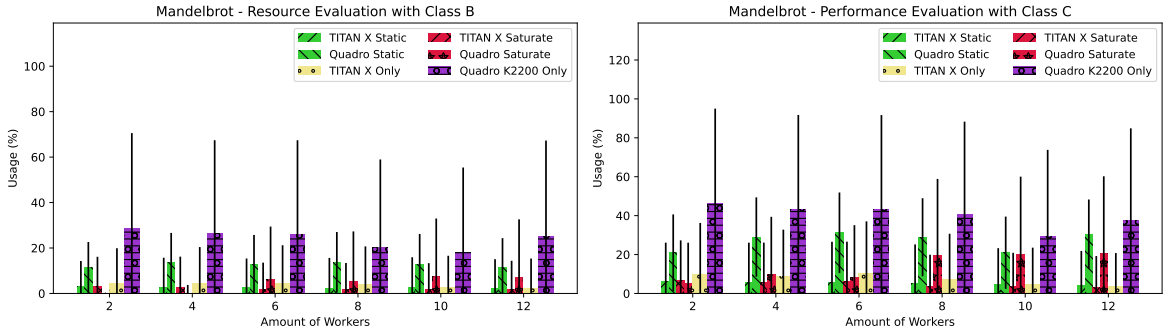

Figure 5.24 – Mandelbrot with Batching and On-Demand - Resource Consumption Evaluation

5.3 Results implementing into OpenMP

In order to evaluate the performance and overhead of our approaches against other possible solutions, we chose to also implement the tested applications utilizing OpenMP annotations in conjunction with CUDA; we limited our tests to the LD, MB, and RT benchmarks, seeing as their execution flowchart are small and very easy to be ported. Based on the above tests done using our policies and the fact that *Saturate* was not able to be fully utilized in most applications, we focused on bringing the *Static* policy into our OpenMP+CUDA implementation of the applications using multi-GPU. The throughput and resource consumption of these tests are shown in Figures 5.25 to 5.30. It is important to note that although SPar can generate OpenMP code, we chose not to utilize the generated code and opted for manual implementation.

Despite its small loads for the GPU and high quantity of communications between the CPU and the GPU, the LD benchmark achieved a much higher throughput than any of our tests using GSParLib. It showed higher usage of resources, meaning that the tasks were more efficiently passed to the GPUs when using the CUDA library than the abstractions provided by GSParLib, as well as better scalability, growing its throughput and usage when more workers were used, this is especially seem in the *Static* policy using workload

C, while the single-GPU executions stagnate their throughput, the *Static* can continue to grow since it is not limited by the resources contained in a single GPU.

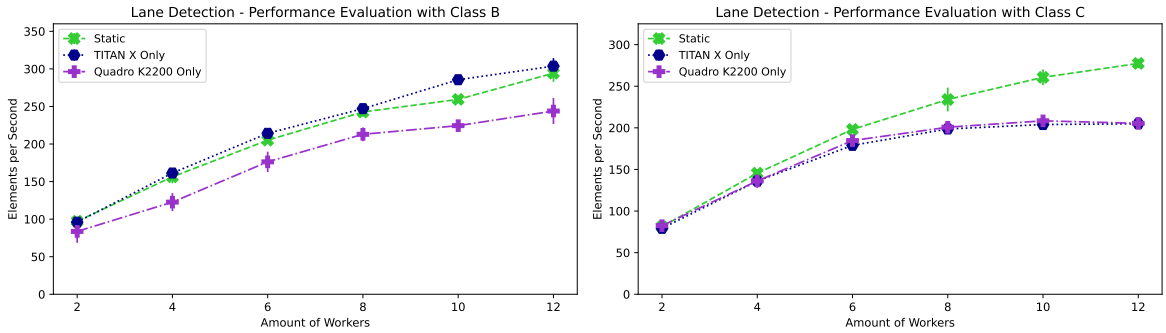

Figure 5.25 – LaneDetection OpenMP - Throughput Evaluation

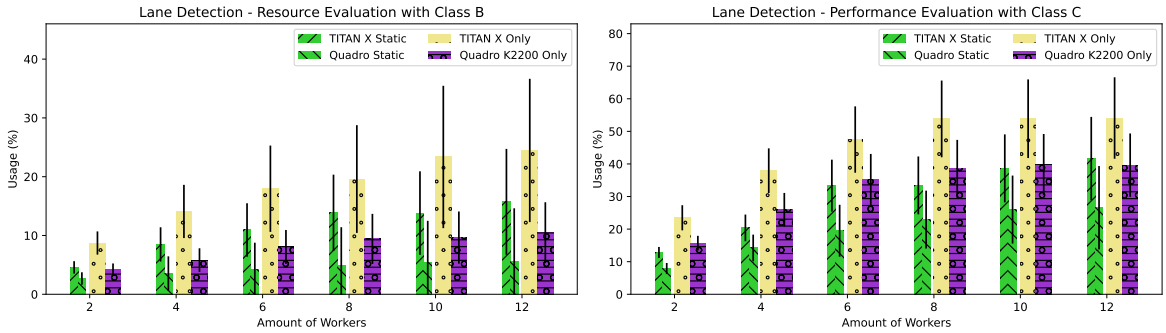

Figure 5.26 – LaneDetection OpenMP - Resource Consumption Evaluation

The MB benchmark, when run using the OpenMP+CUDA, shows its simplicity since it generates even smaller loads for the GPU than the LD benchmark, its results are highly similar in both workloads and uses a fraction of the resources used when run using the SPar+GSParLib implementations, be it in single or multi-GPU modes.

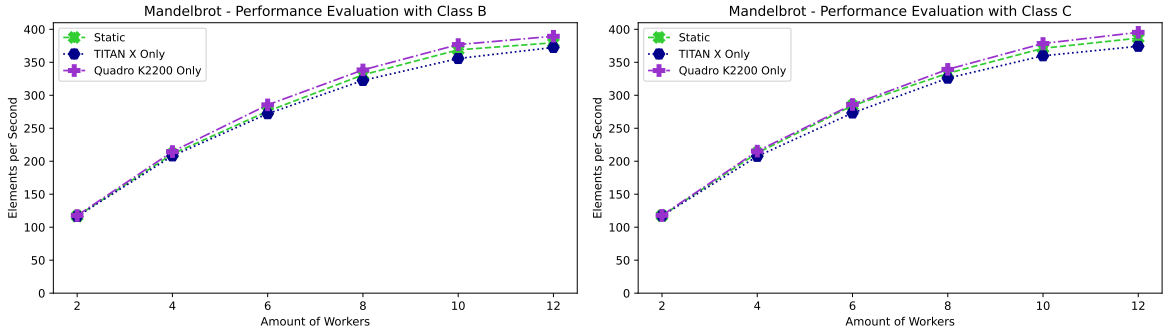

Figure 5.27 – Mandelbrot OpenMP - Throughput Evaluation

Assessing the RT benchmark using OpenMP+CUDA in comparison to the SPar implementation (Figures 5.11 and 5.19), we can see that OpenMP+CUDA presents similar results to the execution utilizing the on-demand scheduler of FastFlow (Figure 5.19), although with both single-GPU executions being close in results. We can also see that the

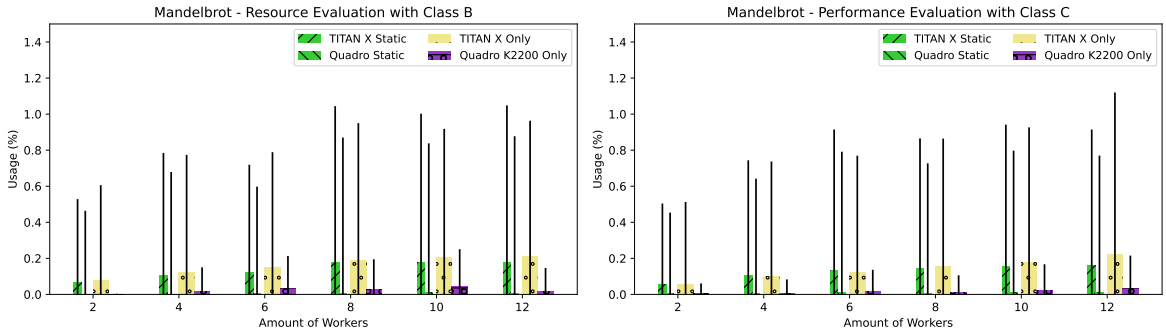

Figure 5.28 – Mandelbrot OpenMP - Resource Consumption Evaluation

Static policy was able to better scale when used on OpenMP+CUDA, probably due to the elimination of the abstraction layer that is present when running the kernel code in GSParLib.

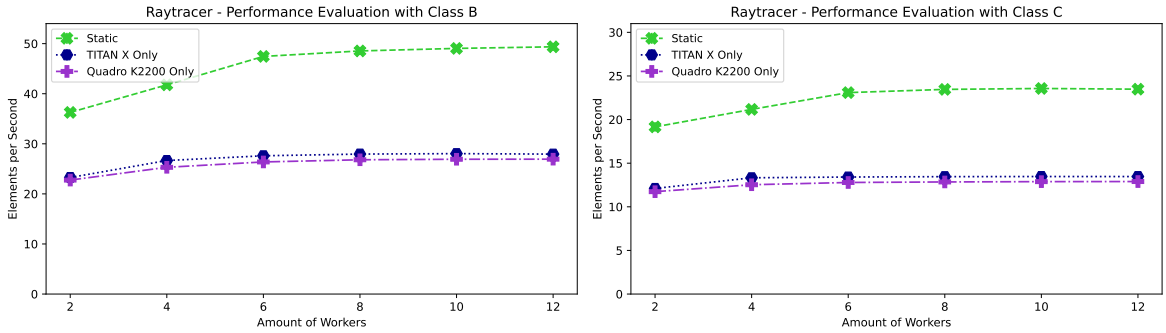

Figure 5.29 – Raytracer OpenMP - Throughput Evaluation

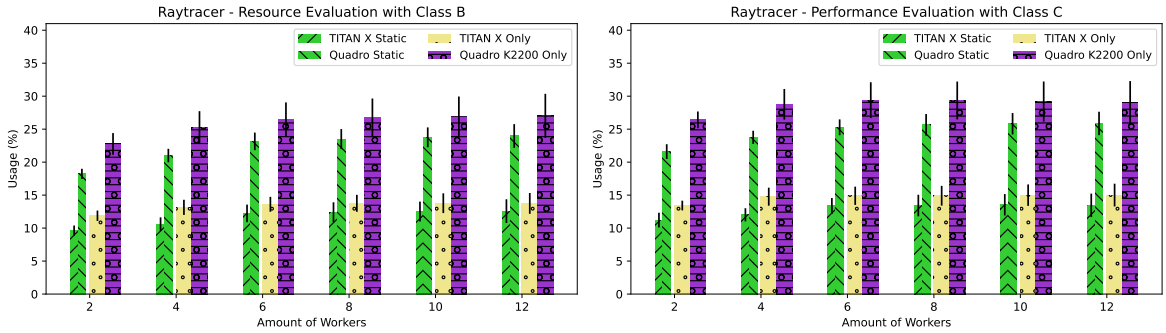

Figure 5.30 – Raytracer OpenMP - Resource Consumption Evaluation

Comparing the benchmark on OpenMP+CUDA to the implementation in SPar we can see that the results obtained by using GSParLib benefit more from the single-GPU executions, with these executions normally achieving the highest throughput when using the Titan X, in contrast, the OpenMP+CUDA seems to efficiently use the GPUs, giving similar results independent of which is chosen to run the code, which in turn benefits our policies, since they seek to use multiple GPUs in an efficient manner.

5.4 Overhead Evaluation

In this section, we will compare the performance of the implemented policies in SPar, Table 5.1 summarizes the results by presenting the performance improvement of the usage of multi-GPU against single-GPU in SPar. The first column indicates the benchmark application. The second indicates the workload used. The third indicates the metric measured. The fourth column indicates the percentage of performance improvement of the *Static* policy over the usage of only the Titan X. The fifth column indicates the percentage of performance improvement of the *Saturate* policy over the usage of only the Titan X.

Based on the results obtained, the *Static* policy offers the best improvements, at least in our tested environment, *Saturate* loses a lot of power since it needs to regularly query the current usage and available memory to decide which GPU should be used, as well as when targeting the other GPUs on the environments needing some time to return the send of tasks to the first GPU on the environment, which contributes to its loses.

Table 5.1 – Performance improvement of multi-GPU over single-GPU

Benchmark	Workload	Metric	Static improve over Titan X	Saturate improve over Titan X
AR	B	Throughput	10.99%	0.20%
AR	C	Throughput	7.10%	-0.05%
LD	B	Throughput	-2.42%	-30.47%
LD	C	Throughput	-13.88%	-31.33%
MB	B	Throughput	1.37%	-33.08%
MB	C	Throughput	-2.20%	-31.93%
RT	B	Throughput	2.36%	-6.43%
RT	C	Throughput	8.57%	-2.22%

Table 5.2 summarizes the results by presenting the performance improvement of the usage of the default Round-robin scheduling against the On-demand scheduling, both part of the FastFlow backend utilized by SPar. The first column indicates the benchmark application. The second indicates the workload used. The third indicates the metric measured. The fourth column indicates the percentage of performance improvement of the On-demand scheduler over the usage of the Round-robin scheduler.

Table 5.2 only takes into account the results regarding the *Static* policy since only it obtained improvements when using the on-demand scheduler, these results are mainly because of an equal division of work between both GPUs, as done when using the round-robin scheduler, is detrimental to the runtime, since the second GPU, Quadro K2200, has less computational power, it takes more time to process the tasks which leads to a backlog of tasks on the workers targeting this GPU.

Table 5.3 summarizes the results by presenting the performance improvement of the usage of the batch grouping of tasks provided by SPar when utilizing GPU. The first col-

Table 5.2 – Performance improvement of On-demand scheduling over Round-robin scheduling

Benchmark	Workload	Metric	On-demand improve over Round-robin
AR	B	Throughput	7.19%
AR	C	Throughput	34.04%
LD	B	Throughput	235.13%
LD	C	Throughput	269.31%
MB	B	Throughput	22.39%
MB	C	Throughput	21.87%
RT	B	Throughput	130.06%
RT	C	Throughput	96.59%

umn indicates the benchmark application. The second indicates the workload used. The third indicates the metric measured. The fourth column indicates the percentage of performance improvement of the batch optimization over the default stream parallelism. The fifth column indicates the percentage of performance improvement of the batch optimization in conjunction with the On-demand scheduler over the default stream parallelism.

The results show that the *Static* policy is the most improved when using the batch grouping, this is mainly due to the reduced amount of transfers done when utilizing such optimization to run the application. Utilizing on-demand in conjunction with batch optimization does not add any significant improvements, having a reduced throughput over simply utilizing the on-demand scheduler. *Saturate* as a whole did not scale with the batch optimization, mainly due to the grouping aspect of the optimization.

Table 5.3 – Performance improvement of batch over non-batch execution

Benchmark	Workload	Metric	Static/Batch improve	Static/Batch On-demand improve	Saturate/Batch improve	Saturate/Batch On-demand improve
MB	B	Throughput	20.20%	-1.71%	-0,39%	-0,24%
MB	C	Throughput	53.56%	0.25%	-2,02%	-0,96%

Table 5.4 summarizes the results by presenting the performance improvement of the SPar's generated code over the manual implementations, discussed in Chapter 4, and the performance improvement of the OpenMP manual implementation over the manual implementations. The first column indicates the benchmark application. The second indicates the workload used. The third indicates the metric measured. The fourth column indicates the percentage of performance improvement of SPar's generated code over the manual implementations. The fifth column indicates the percentage of performance improvement of the OpenMP codes over the manual implementations.

The results show that for the simpler applications, LD and MB, SPar's generated code suffers a significant loss of throughput, this could be mainly due to the large amount

Table 5.4 – Performance improvement of SPar and OpenMP over manual implementation

Benchmark	Workload	Metric	SPar improve over manual	OpenMP improve over manual
LD	C	Throughput	-62.74%	146.80%
MB	B	Throughput	-53.55%	232.51%
RT	C	Throughput	98.54%	89.15%

of memory transfers necessary to process the streams or the excessive amount of kernel clones done to abstract GSParLib to cover all the possible generations present in SPar.

5.5 Impact on programmability

This section discusses aspects of SPar's programmability compared to the other implementations. Figure 5.31 presents the results for each implemented benchmark. Figure 5.31(a) presents the results for the AR benchmark. Figure 5.31(b) presents the results for the benchmarks LD, MB, and RT. The x-axis lists the benchmarks, and the y-axis presents the number of source lines of code. We used the software SLOC [64] to collect the source lines of code. This metric does not represent code productivity.

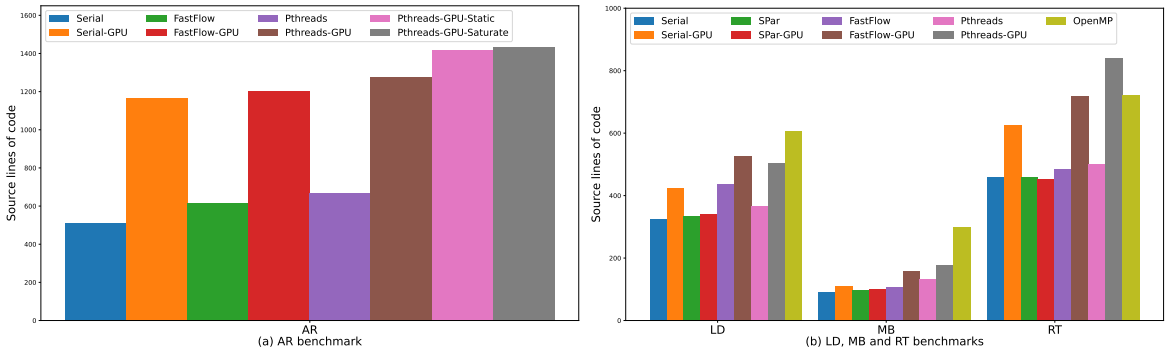

Figure 5.31 – Source Lines of Code of each Benchmark tested.

When it comes to multi-core versions, using Pthread necessitates a manual implementation of each component required for a stream processing application. This includes creating queues for communication between stages, functions for adding elements to the queues, and techniques to manage communications among different threads. In contrast, FastFlow provides algorithmic skeletons specifically designed for programming stream processing applications. This means that users do not have to create queues for stage communications or scheduling mechanisms. However, users still need to implement detailed routines, such as defining C++ classes for the computations of each stage.

SPar significantly enhances programmability compared to Pthread and FastFlow by automatically generating all the routines necessary for a stream application. As a

result, SPar requires less programming effort than FastFlow, which, in turn, requires less effort than Pthread.

The challenges of programming become even greater when dealing with GPUs. SPar's annotations simplify GPU programming by eliminating the need for multiple mechanisms essential for parallelizing code. This allows for straightforward CPU parallelism combined with GPU utilization. Consequently, the source lines of code in SPar are essentially equivalent to the serial versions of the applications, whereas manually coded versions that utilize both CPU and GPU can require up to three times more lines of code due to the components necessary for stream processing and GPU kernels.

5.6 Final Remarks about Multi-GPU with SPar

In this chapter, we roughly presented the implementation and evaluation of two policies to target multi-GPU via GSParLib's code abstraction, and we evaluated SPar overhead by comparing its performance to a manual implementation using OpenMP combined with CUDA. Our tests highlighted several limitations in the usage of GPU as a whole in SPar, although we do not talk much about them when discussing the results of our tests, most of them were already thoroughly explored on Gabriell's thesis[23], including an excessive amount of GPU kernel copies, and somewhat excessive amount of memory transfers due to the stream parallelism inherent overhead such as allocating memory for each new stream element.

When comparing all our tests as a whole, we can see that the *Static* policy should be used in conjunction with schedulings such as On-Demand or Work Stealing, where the workers paired with the less computationally endowed GPUs do not become overflow by a backlog of tasks while the more potent GPUs are idle.

Our tests also show that the *Saturate* policy seems extremely dependent on the GPUs having similar computational powers, as seen on the executions of the MB benchmarks when targeting the second GPU, the benchmark suffers some loss in exchange for not overloading a GPU. One of the problems that we did not discuss, at least on the tests, is that *Saturate* has an overhead in the sense that the choice of which GPU to use is made based on a consultation with a library, and this could become a bottleneck if done too frequently, as such we limited the consultation to be done in the emitter, which introduces a delay since new works will be sent before the first one began to process and the consultation began to receive data regarding the executions. This could be resolved by executing the consultation on the workers, but this introduces a new problem in that more consultations would be done, which would overwhelm the GPUs and hinder the processing of tasks.

Regarding the usage of batch to help alleviate the inherent overhead present in streaming, only *Static* appears to get some improvement, reaching results akin to the executions using singular GPUs, and bringing the Quadro K2200 up to throughput with the Titan X. Using batch in conjunction with an on-demand scheduling does not bring any improvement, to the contrary, with the workload C the execution targeting only the Quadro K2200 suffers a degradation of throughput if compared to only using the batch approach.

Our tests with OpenMP+CUDA show that there is much improvement to be gained by using multi-GPU, but much improvement could not be achieved in our main tests utilizing SPar with GSParLib. This is due to the limitation imposed by FastFlow or even because the abstractions implemented by GSParLib are hindering the efficient usage of GPUs. More tests could be done in the future seeking to pinpoint where the bottlenecks of execution are present in our tests by reviewing our rough implementations in PThread to make a full comparison or even introducing new implementations, like the OpenMP+CUDA that we used to test if our implementations worked outside of the confines of the SPar implementation.

By analyzing the results and the number of source code lines of different ideas to explore multi-GPU, we can somewhat say that regarding our main research question, **Can C++ annotations, like those provided by SPar for stream parallelism, simplify multi-GPU parallelism exploitation without impacting performance?**, that yes, it is possible to simplify the exploitation of multi-GPU for stream parallelism without sacrificing the performance of the application, although SPar's usage of FastFlow imposes some difficulties due to the overheads present in the FastFlow, especially when targeting GPU problems without much loads, such as MB and LD.

6. CONCLUSION

In this work, we began by seeking possible scheduling policies for multi-GPU usage in stream applications in the current literature. Based on our search, we selected four possible scheduling policies, which were tested and fine-tuned by implementing them into four applications. After the fine-tuning, the policies were abstracted and integrated into SPar's code generation, where more tests were done, with the purpose of evaluating the performance of the policies into SPar, as well as how much overhead impact these policies had when integrated onto SPar execution.

Our tests show that the current environment of SPar and GSParLib already supports the usage of multi-GPUs, although it appears that this support still presents some limitations thanks to the abstractions present in GSParLib. Although our tests using SPar and GSParLib show no improvement when compared to a single-GPU execution of the applications, by combining our approaches with methods already present in FastFlow, we achieved equal results with less consumption of resources. Our tests utilizing OpenMP and CUDA, also show that our approaches have more scalability, being able to surpass the single-GPU executions and also showing that the implementations utilizing SPar and GSParLib were not able to fully extract the computational power present in the environment, be it because of FastFlow's code limitations when processing streams or the excessive amount of transfers and copies done to allow the stream parallelism done in the applications.

We summarize the main insights and achievements of our tests as follows: 1) Multi-GPU methodologies for easy usage inside SPar; 2) Evaluation of how the current SPar environment merges with our multi-GPU methodologies; 3) Comparison of one of our methodologies against a state-of-the-art framework that supports multi-GPU.

GSParLib and SPar stand out with unique characteristics among various frameworks in industry and academia. GSParLib fully supports a unified interface for standard APIs in GPU programming. The system's mechanisms for utilizing stream parallelism are seamlessly integrated for users. It automatically handles mutual exclusion during GPU memory allocation and deallocation, along with the creation and management of CUDA streams and OpenCL command queues to enable asynchronous GPU kernel execution. GSParLib employs standard C/C++ containers, eliminating the need for a new data type, and works with standard C/C++ compilers like g++, without requiring a separate compiler. Meanwhile, SPar distinguishes itself by using code annotations to generate code for stream processing targeting CPU and GPU, effectively combining stream and data parallelism.

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

Deeply evaluate and improve the support for multi-GPU on GSParLib: Currently, GSParLib allows the user to allocate memory or launch a GPU kernel by specifying the GPU ID that they desire to use. However, there is currently no deep evaluation of the possible overhead in the internal mechanisms of GSParLib when using multiple GPUs. Another point is that GSParLib should be able to distribute the computations of a parallel pattern among several GPUs, but is currently only able to assign the computations to a single GPU.

Provide GPU runtime optimizations to SPar: Much like our *Saturate* policy, future improvements could focus on GPU optimizations for SPar during execution. For example, SPar might autonomously adjust the number of threads per block while processing a GPU kernel, selecting optimal configurations to handle new stream elements. In the current state of SPar, even with the usage of our policies, the GPUs can accept new stream elements even if there is not sufficient device memory available, causing the executions to fail. Moreover, there are still instances where CPU threads retain copies of GPU kernels until the application completes, even though SPar tries to free up memory by deleting specific GPU kernels it no longer needs, sometimes this is not done correctly. These optimizations are essential for enhancing SPar's performance and adaptability in managing various stream applications that require substantial data processing.

Provide multi-GPU runtime optimizations to GSParLib: Although GSParLib supports multi-GPU, its usage is limited to only the setting in which GPU the kernel code is going to be run. Future work could improve the runtime of GSParLib by targeting multi-GPU communication technologies, such as NVLink and NVSwitch, which would allow a broader usage of multi-GPU and the processing of more robust benchmarks.

Evaluate the performance of multi-GPU on distributed environments: Multi-GPU is a broad term, varying from single-node multi-GPU to a distributed multi-GPU per node environment, our work focused only on the single-node aspect of multi-GPU, but in a future work, the distributed side of multi-GPU could be evaluated, to do this, GSParLib would need to receive some improvements, mainly targeting the single-node nature of its implementations. This would also embrace multi-GPU evaluations using GPUs belonging to different vendors, such as NVIDIA and AMD.

Evaluate the performance of multi-GPU on other accelerators: Although GSParLib supports only GPUs, the framework could support other many-core accelerators such as Field Programmable Gate Arrays (FPGA), Intel Many Integrated Core (MIC). Like GPUs from AMD manufacture, the OpenCL can be used to exploit parallelism on other many-core accelerators. This was not touched on in our tests; it could open a wide range of tests regarding different accelerators.

Provide new parallel patterns to GSParLib and test their usability for multi-GPU: There are still a good amount of parallel patterns not implemented in GSParLib, such as gather, stencil, scan, and scatter; if these patterns were available in GSPar-

Lib's Pattern API, then the programmer would not need to implement a variation of the map parallel pattern to use those functionalities, requiring less programming effort. Although these patterns are mainly for data parallelism, it would also be essential to support streaming-related features such as thread safety and batching when implementing these patterns to allow their usage in possible stream parallelism applications.

Provide new scheduling approaches for multi-GPU on SPar and GSParLib: In this study, we presented two scheduling methods for multi-GPU usage within SPar, utilizing GSParLib, and examined how existing task distribution methods in SPar interact with our multi-GPU strategies. Future research could explore either the avenue of integrating novel work scheduling methods into SPar or developing new multi-GPU scheduling techniques based on other frameworks found in current literature. Additionally, future projects might adapt our approaches to integrate with other frameworks supported by SPar, such as OpenMP.

Automatic and hybrid parallelism support in SPar: In its current form, the SPar compiler always offloads pure code regions to the GPU accelerator when set with the corresponding flag. Future works may explore algorithms to automatically choose if such offloading would present improvements to the execution. They may also choose the best scheduling approaches by analyzing the code characteristics and querying all the devices' properties present in the environment.

REFERENCES

[1] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-Level and Efficient Streaming on Multicore”. 2014.

[2] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-Level and Efficient Streaming on Multicore”. John Wiley & Sons, Ltd, 2017, chap. 13, pp. 261–280.

[3] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores”. In: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2021), 2021, pp. 291–295.

[4] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A parallel programming assessment for stream processing applications on multi-core systems”, *Computer Standards & Interfaces*, vol. 84, March 2023, pp. 103691.

[5] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing”. In: 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022), 2022, pp. 229–232.

[6] Andrade, H.; Gedik, B.; Turaga, D. S. “Fundamentals of stream processing: application design, systems, and analytics”. Cambridge, England: Cambridge University Press, 2014.

[7] Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P.-A. “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures”, *Concurrency and Computation: Practice and Experience*, vol. 23–2, 2011, pp. 187–198, https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631.

[8] Banger, R.; Bhattacharyya, K. “OpenCL programming by example”. Birmingham, England: Packt Publishing, 2013.

[9] Board, O. A. R. “Openmp application programming interface version 5.2”. Source: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, Jan 2024.

[10] Chakravarthy, S.; Jiang, Q. “Stream Data Processing: A Quality of Service Perspective. Modeling, Scheduling, Load Shedding, and Complex Event Processing”. New York, NY: Springer, 2009.

[11] Chandrasekaran, S.; Juckeland, G. “OpenACC for Programmers”. Boston, MA: Addison-Wesley Educational, 2017.

[12] Cheng, J.; Grossman, M.; McKercher, T. “Professional CUDA C Programming”. Indianapolis, IN: Wrox Press, 2014.

[13] Cole, M. “Algorithmic skeletons: structured management of parallel computation”. Cambridge, MA, USA: MIT Press, 1991.

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

[15] Cook, S. “CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs”. Cambridge, MA: Elsevier Science, 2013.

[16] Corporation, A. “Hip”. Source: https://github.com/ROCm/HIP, Jan 2024.

[17] Corporation, N. “Sli best practices”. Source: http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf, Jan 2024.

[18] Corporation, N. “Nvidia nvswitch – the world’s highest-bandwidth on-node switch”. Source: http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, Jan 2024.

[19] Corporation, N. “Cuda c programming guide”. Source: https://docs.nvidia.com/cuda/archive/10.1/pdf/CUDA_C_Programming_Guide.pdf, August 2019.

[20] Corporation, N. “Nvidia hpc compilers”. Source: https://docs.nvidia.com/hpc-sdk/archive/20.7/pdf/hpc207c++_par_alg.pdf, August 2020.

[21] Corporation, N. “Cuda c++ programming guide”. Source: https://docs.nvidia.com/cuda/archive/12.2.1/pdf/CUDA_C_Programming_Guide.pdf, Jan 2024.

[22] Corporation, N. “Nvidia hpc sdk”. Source: https://developer.nvidia.com/hpc-sdk, August 2023.

[23] de Araujo, G. A. “Data and stream parallelism optimizations on gpus”, Master’s Thesis, Porto Alegre, RS, Brasil, 2022, 113p.

[24] De Matteis, T.; Mencagli, G. “Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing”. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016.

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

[26] Ernstsson, A.; Ahlqvist, J.; Zouzoula, S.; Kessler, C. “Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters”, *International Journal of Parallel Programming*, vol. 49–6, May 2021, pp. 846–866.

[27] Foley, D.; Danskin, J. “Ultra-performance pascal gpu and nvlink interconnect”, *IEEE Micro*, vol. 37–2, 2017, pp. 7–17.

[28] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[29] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “An Embedded C++ Domain-Specific Language for Stream Parallelism”. In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, 2015, pp. 317–326.

[30] 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, March 2017, pp. 1740005.

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

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

[33] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: XXIV Brazilian Symposium on Programming Languages (SBLP), 2020, pp. 48–55.

[34] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “OpenMP as runtime for providing high-level stream parallelism on multi-cores”, *The Journal of Supercomputing*, vol. 78–1, January 2022, pp. 7655–7676.

[35] Kim, G.; Lee, M.; Jeong, J.; Kim, J. “Multi-gpu system design with memory networks”. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 484–495.

[36] Kirk, D. B.; Hwu, W.-M. W. “Programming Massively Parallel Processors”. Cambridge, MA: Elsevier Science, 2017, 576p.

[37] Le, D. T.; Nguyen, H. D.; Pham, T. A.; Ngo, H. H.; Nguyen, M. T. “An intermediate library for multi-gpus computing skeletons”. In: 2012 IEEE RIVF International Conference on

Computing; Communication Technologies, Research, Innovation, and Vision for the Future, 2012.

[38] Li, H.; Tarik, K.; Arefnezhad, S.; Magosi, Z. F.; Wellershaus, C.; Babic, D.; Babic, D.; Tihanyi, V.; Eichberger, A.; Baunach, M. C. “Phenomenological modelling of camera performance for road marking detection”, *Energies*, vol. 15–1, 2022.

[39] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for Multi-Cores”. In: XXV Brazilian Symposium on Programming Languages (SBLP), 2021, pp. 41–48.

[40] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “Combining stream with data parallelism abstractions for multi-cores”, *Journal of Computer Languages*, vol. 73, December 2022, pp. 101160.

[41] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “DSParLib: A C++ Template Library for Distributed Stream Parallelism”, *International Journal of Parallel Programming*, vol. 50–5, 2022, pp. 454–485.

[42] Matsumura, K.; Sato, M.; Boku, T.; Podobas, A.; Matsuoka, S. “MACC: An OpenACC Transpiler for Automatic Multi-GPU Use”. Springer International Publishing, 2018, pp. 109–127.

[43] Matz, A.; Doerfert, J.; Fröning, H. “Automated partitioning of data-parallel kernels using polyhedral compilation”. In: Workshop Proceedings of the 49th International Conference on Parallel Processing, 2020.

[44] McCool, M.; Reinders, J.; Robison, A. “Structured Parallel Programming: Patterns for Efficient Computation”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, 1st ed..

[45] Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. “OpenCL Programming Guide”. Boston, MA: Addison-Wesley Educational, 2011.

[46] 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: 2014 First Workshop on Accelerator Programming using Directives, 2014.

[47] OpenACC-Standard.org. “The openacc application programming interface”. Source: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf, Jan 2024.

[48] Pabst, S.; Koch, A.; Straßer, W. “Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces”, *Computer Graphics Forum*, vol. 29–5, 2010, pp. 1605–1612, https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01769.x.

[49] Pieper, R. L. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 170p.

[50] Rasch, A.; Bigge, J.; Wrodarczyk, M.; Schulze, R.; Gorlatch, S. “docal: high-level distributed programming with opencl and cuda”, *The Journal of Supercomputing*, vol. 76–7, March 2019, pp. 5117–5138.

[51] Rep., K. G. T. “Sycl 2020 specification (revision 8)”. Source: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, Jan 2024.

[52] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 163p.

[53] Rockenbach, D. A.; Araujo, G.; Griebler, D.; Fernandes, L. G. “GSParLib: A multi-level programming interface unifying OpenCL and CUDA for expressing stream and data parallelism”, *Computer Standards & Interfaces*, vol. 92, March 2025, pp. 103922.

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

[55] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for GPUs”. In: XXVI Brazilian Symposium on Programming Languages (SBLP), 2022, pp. 41–49.

[56] Sanders, J.; Kandrot, E. “CUDA by example: An Introduction to General-Purpose GPU Programming”. Boston, MA: Addison-Wesley Educational, 2010.

[57] Schaetz, S.; Uecker, M. “A Multi-GPU Programming Library for Real-Time Applications”. Springer Berlin Heidelberg, 2012, pp. 114–128.

[58] Steuwer, M.; Kegel, P.; Gorlatch, S. “Towards high-level programming of multi-gpu systems using the skelcl library”. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops; PhD Forum, 2012.

[59] Thies, W.; Amarasinghe, S. “An empirical characterization of stream programs and its implications for language and compiler design”. In: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 365–376.

[60] Trott, C. R.; Lebrun-Grandie, D.; Arndt, D.; Ciesko, J.; Dang, V.; Ellingwood, N.; Gayatri, R.; Harvey, E.; Hollman, D. S.; Ibanez, D.; Liber, N.; Madsen, J.; Miles, J.; Poliakoff, D.;

Powell, A.; Rajamanickam, S.; Simberg, M.; Sunderland, D.; Turcksin, B.; Wilke, J. “Kokkos 3: Programming model extensions for the exascale era”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–4, April 2022, pp. 805–817.

[61] 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”, *Softw. Pract. Exper.*, vol. 40–12, nov 2010, pp. 1073–1104.

[62] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing High-Level Self-Adaptive Abstractions for Stream Parallelism on Multicores”, *Software: Practice and Experience*, vol. 51–6, January 2021, pp. 1194–1217.

[63] Wang, Z.; Li, P.; Hou, R.; Li, Z.; Cao, J.; Wang, X.; Meng, D. “He-booster: An efficient polynomial arithmetic acceleration on gpus for fully homomorphic encryption”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 34–4, 2023, pp. 1067–1081.

[64] Wheeler, D. A. “Sloccount”. Source: https://dwheeler.com/sloccount/, Feb 2025.

[65] Wilt, N. “CUDA handbook”. Boston, MA: Addison-Wesley Educational, 2013.

[66] Wrede, F.; Kuchen, H. “Towards high-performance code generation for multi-gpu clusters based on a domain-specific language for algorithmic skeletons”, *International Journal of Parallel Programming*, vol. 48–4, May 2020, pp. 713–728.

[67] Xu, Q.; Jeon, H.; Annavaram, M. “Graph processing on gpus: Where are the bottlenecks?” In: 2014 IEEE International Symposium on Workload Characterization (IISWC), 2014, pp. 140–149.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br