PUCRS

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

GABRIELL ALVES DE ARAUJO

DATA AND STREAM PARALLELISM OPTIMIZATIONS ON GPUS

Porto Alegre
2022

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

DATA AND STREAM PARALLELISM OPTIMIZATIONS ON GPUS

GABRIELL ALVES DE ARAUJO

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

Porto Alegre
2022

Ficha Catalográfica

A663d Araujo, Gabriell Alves de

Data and Stream Parallelism Optimizations on GPUs / Gabriell Alves de Araujo. – 2022.
111 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Coorientador: Prof. Dr. Dalvan Jair Griebler.

1. Parallel Programming. 2. GPU Programming. 3. Heterogeneous Computing. 4. Data Parallelism. 5. Stream Parallelism. I. Fernandes, Luiz Gustavo Leão. II. Griebler, Dalvan Jair. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Loiva Duarte Novak CRB-10/2079

GABRIELL ALVES DE ARAUJO

DATA AND STREAM PARALLELISM OPTIMIZATIONS ON GPUS

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 31, 2022.

COMMITTEE MEMBERS:

Prof. Dr. Horacio González-Vélez (National College of Ireland)

Prof. Dr. Tiago Ferreto (PPGCC/PUCRS)

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

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

OTIMIZAÇÕES DO PARALELISMO DE STREAM E DADOS EM GPUS

RESUMO

Nos dias de hoje, a maioria dos computadores são equipados com unidades de processamento gráfico (GPUs) para prover capacidade massiva de paralelismo a baixo custo. Para explorar completamente a capacidade destas arquiteturas é necessário o uso de programação paralela. No entanto, isso representa um desafio para os programadores, pois requer a refatoração de algoritmos, técnicas de paralelismo e conhecimento específico sobre o hardware. Além disso, o paralelismo das GPUs é ainda mais desafiador, pois GPUs possuem características de hardware peculiares, e outro paradigma de paralelismo chamado de programação *many-core*. Nesse sentido, pesquisas de computação paralela tem se concentrado no estudo de técnicas de programação eficientes para GPUs, e também no desenvolvimento de abstrações que diminuem o esforço de programação. SPar é uma linguagem específica de domínio (DSL) que segue essa direção de pesquisa. Programadores podem utilizar a SPar para expressar paralelismo de *stream* sem impactar significativamente o desempenho. A SPar oferece abstrações de alto nível através de anotações no código-fonte, enquanto o compilador da SPar gera código paralelo. Recentemente foi adicionada uma extensão na SPar, a qual permite que seja gerado código paralelo para CPUs e GPUs em aplicações de stream. Os núcleos da CPU controlam o fluxo de dados, e a GPU aplica paralelismo massivo na computação de cada elemento do fluxo de dados. Para este fim, a SPar gera código para uma biblioteca intermediária chamada GSParLib. GSParLib é uma API paralela orientada a padrões que provê um modelo único de programação para a *runtime* dos *frameworks* CUDA e OpenCL, permitindo a exploração do paralelismo em GPUs de diferentes fabricantes. Porém, o suporte para GPUs em ambas SPar e GSParLib ainda está em seus passos iniciais; SPar e GSParLib oferecem apenas funcionalidades básicas, e nenhum estudo avaliou o desempenho de forma abrangente. A contribuição deste trabalho concentra-se em paralelizar benchmarks representativos da área de computação de alto-desempenho (HPC), fornecer novos recursos e otimizações para GPUs na

GSParLib e SPar, e apresentar um método para prover *frameworks* que sejam agnósticos e independentes de interfaces de programação de baixo nível. O conjunto de melhorias cobre a maioria das limitações críticas de desempenho e programabilidade da GSParLib. Nos experimentos deste trabalho, a versão otimizada da GSParLib foi capaz de atingir até 54.500,00% de melhoria no *speedup* em relação à versão original da GSParLib nos benchmarks de paralelismo de dados e até 718,43% de melhoria no *throughput* nos benchmarks de paralelismo de *stream*.

Palavras-Chave: Programação Paralela, programação de GPUs, computação heterogênea, paralelismo de dados, paralelismo de stream, programação paralela estruturada, padrões paralelos, benchmarks, aplicações de processamento de stream, linguagem específica de domínio, esqueletos algorítmicos, avaliação de desempenho, computação de alto desempenho, C, C++, CUDA, OpenCL.

DATA AND STREAM PARALLELISM OPTIMIZATIONS ON GPUS

ABSTRACT

Nowadays, most computers are equipped with Graphics Processing Units (GPUs) to provide massive-scale parallelism at a low cost. Parallel programming is necessary to exploit this architectural capacity fully. However, it represents a challenge for programmers since it requires refactoring algorithms, designing parallelism techniques, and hardware-specific knowledge. Moreover, GPU parallelism is even more challenging since GPUs have peculiar hardware characteristics and employ a parallelism paradigm called many-core programming. In this sense, parallel computing research has focused on studying efficient programming techniques for GPUs and developing abstractions that reduce the effort when writing parallel code. SPar is a domain-specific language (DSL) that goes in this direction. Programmers can use SPar to express stream parallelism in a simpler way without significantly impacting performance. SPar offers high-level abstractions via code annotations while the SPar compiler generates parallel code. SPar recently received an extension to allow parallel code generation for CPUs and GPUs in stream applications. The CPU cores control the flow of data in the generated code. At the same time, the GPU applies massive parallelism in the computation of each stream element. To this end, SPar generates code for an intermediate library called GSParLib, a pattern-oriented parallel API that provides a unified programming model targeting CUDA and OpenCL runtime, allowing parallelism exploitation of different vendor GPUs. However, the GPU support for both SPar and GSParLib is still in its initial steps; they provide only basic features, and no studies have comprehensively evaluated SPar and GSParLib's performance. This work contributes by parallelizing representative high-performance computing (HPC) benchmarks, implementing new features and optimizations for GPUs in the GSParLib and SPar, and presenting a method for providing agnostic frameworks independent of low-level programming interfaces. Our set of improvements covers most of the critical limitations of GSParLib regarding performance and programmability. In our experiments, the optimized version of GSParLib

achieved up to 54,500.00% of speedup improvement over the original version of GSParLib on data parallelism benchmarks and up to 718,43% of throughput improvement on stream parallelism benchmarks.

Keywords: Parallel programming, GPU programming, heterogeneous computing, data parallelism, stream parallelism, structured parallel programming, parallel patterns, benchmarks, stream processing applications, domain-specific language, algorithmic skeletons, performance evaluation, high performance computing, C, C++, CUDA, OpenCL.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AHP – Automatic Heterogeneous Pipelining

API – Application Programming Interface

AST – Abstract Syntax Tree

BT – Block Tri-diagonal Solver

CFD – Computational Fluid Dynamics

CG – Conjugate Gradient

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

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DSL – Domain Specific Language

EP – Embarrassingly Parallel

FPGA – Field Programmable Gate Array

FT – Discrete 3D Fast Fourier Transform

GMAP – Parallel Applications Modeling Group

GCC – GNU Compiler Collection

GPU – Graphics Processing Unit

GSPARLIB – GPU Stream Parallelism Library

HPC – High Performance Computing

IS – Integer Sort

LD – Lane Detection

LU – Lower-Upper Gauss-Seidel Solver

MB – Mandelbrot

MG – Multi-Grid

MIC – Many Integrated Core

MPI – Message Passing Interface

MS – Military Server Benchmark

NAS – NASA Advance Supercomputing

NPB – NASA Advance Supercomputing Parallel Benchmarks

OPENACC – Open Accelerators

OPENCL – Open Computing Language

OPENMP – Open Multi-Processing

SLOC – Source Lines of Code

SIMD – Single Instruction Multiple Data

SIMT – Single Instruction Multiple Threads

SM – Stream Multiprocessor

SP – Scalar Penta-diagonal Solver

SPAR – Stream Parallelism

STL – Standard Template Library

RT – Raytracing

XACC – XcalableACC

XMP – XscalableMP

XMP-ACC – XscalableMP-ACC

CONTENTS

1. INTRODUCTION

In the 1980s and early 1990s, operating systems with graphical interfaces such as Microsoft Windows helped create a market for a new type of processor targeting graphics processing. At the same time, the company Silicon Graphics popularized graphics applications such as scientific and technical visualization. In mid-1990, the requirement for progressive computing power to process 3D graphics increased substantially, driven by the industry of games launching games such as Doom, Duke Nukem 3D, and Quake [KH10]. Companies like NVIDIA, ATI Technologies, and 3dfx Interactive started releasing the first graphics accelerators to supply that demand. The first graphics accelerators were single-chip processors with features such as integrated transform, lightning, and rendering engines [SK10]. Figure 1.1 illustrates a summary of the GPU timeline. White rectangles represent milestones. Rectangles with light orange or light yellow backgrounds are important events.

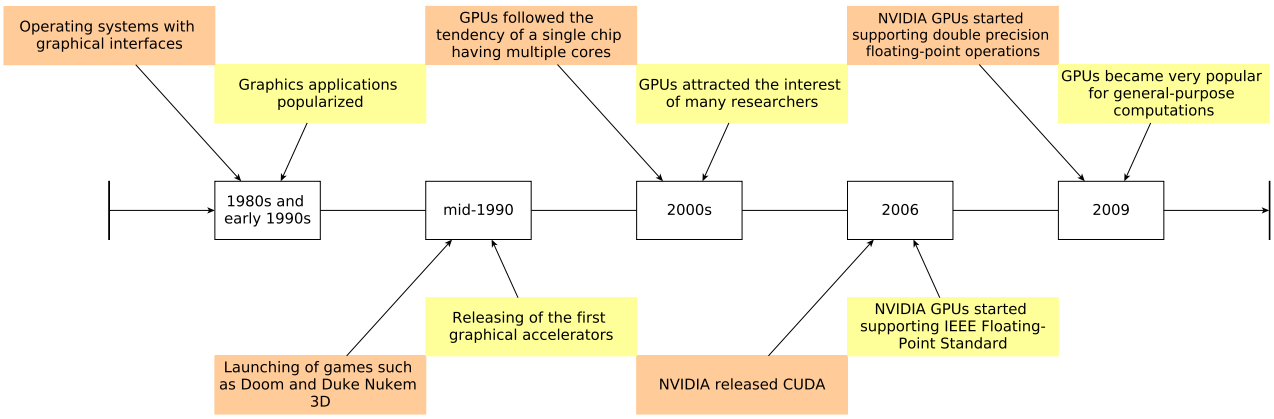


Figure 1.1 – GPU timeline.

Before the 2000s, processors had a single processing core per chip, and the strategy used by the semiconductor industry to improve the performance was to increase the clock speed. However, increasing the processor's clock speed leads to higher temperatures and imposes a significant drawback because the chips cannot dissipate such temperatures. Due to this and other limitations, the industry started to increase the number of processing cores bundled in each chip to keep improving the total performance of processors [Kum02].

During the 2000s, Graphics accelerators, likewise known as Graphics Processing Units (GPUs), also adopted the multiple cores per chip tendency. However, they diverged from traditional Central Processing Units (CPUs) manufacturing in two main ways: 1) they were designed with more cores (called many-core architectures); 2) they were designed with simpler control units. The manufacturers projected these new features to improve the computation of the graphic applications in favor of the older single-core GPUs with integrated rendering. From this point onward, the parallel capacity of the GPUs attracted the interest

of many researchers investigating the feasibility of employing GPUs in different types of applications [SK10].

Until 2005 programming GPUs to perform general-purpose computation was very difficult because it was necessary to use graphic APIs such as OpenGL to access the GPU cores. Then, in 2006, NVIDIA released the CUDA programming model, which allowed programmers to use traditional languages such as C to exploit the GPUs for general-purpose computations. In 2006 NVIDIA GPUs also started to support IEEE Floating-Point Standard, enabling predictable results, and in 2009 they started to support double-precision floating-point operations. Those features led the GPUs to become very popular for general-purpose computing.

Figure 1.2 summarizes a panorama of the GPUs nowadays. The left side of Figure 1.2 lists modern computer architectures that include GPUs as co-processors. The right side of Figure 1.2 lists a set of applications that benefit from such accelerators [NVI22]. The features present in GPUs enabled them to solve problems from various domains that were not viable before, such as deep learning and autonomous vehicles [SK10]. While the list of GPU-accelerated applications continuously increases, GPUs are present in almost every modern computer architecture, from embedded systems and personal computers to the world's top 500 most powerful supercomputers [KH10].

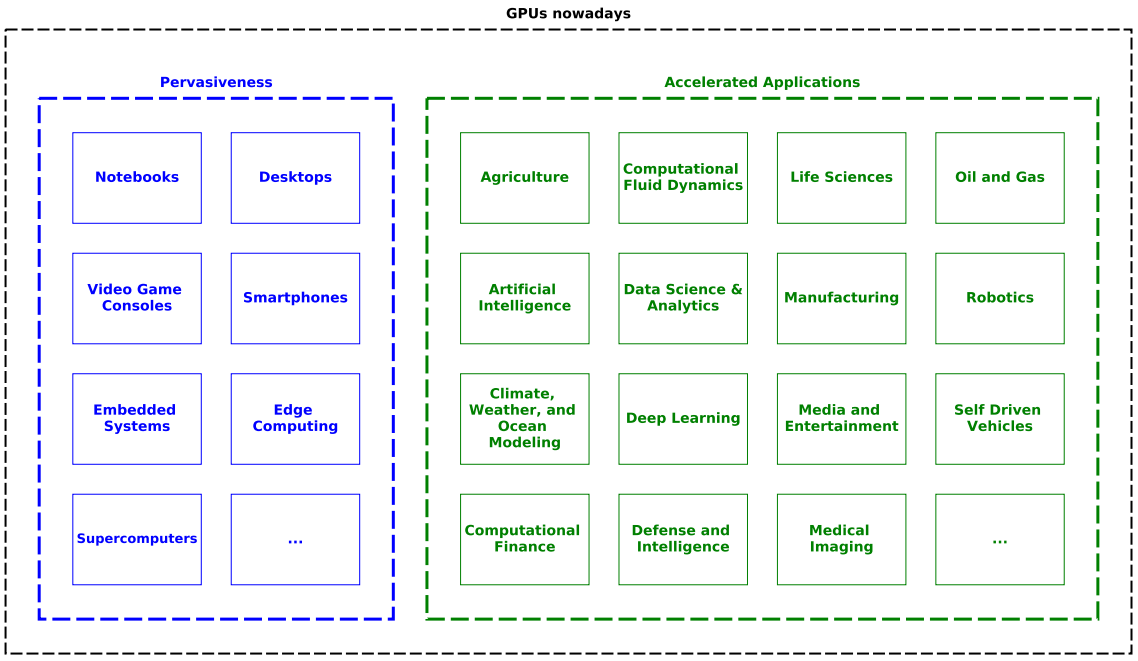


Figure 1.2 – GPUs nowadays, pervasiveness versus accelerated applications.

Many applications that benefit from GPUs' massive parallelism are stream processing applications. Stream processing applications commonly process a continuous flow of data, in which the amount of data may not be known. These kinds of applications are widespread because digital data is very prevalent. Different sources such as social media,

online shopping, and sensors from embedded systems generate lots of data that need to be processed continuously. However, stream processing applications often impose real-time processing constraints to offer relevant results [AGT14].

In order to fulfill the performance requirements of stream processing applications, it is necessary to efficiently exploit the parallel capacity of the hardware on which the application is running. Programmers may do that by employing a parallel programming model. In parallel programming, the programmer rewrites the serial code splitting the program into smaller problems that can be solved concurrently. Nonetheless, writing efficient parallel code is challenging because it requires deep knowledge of the application, hardware, and parallelism techniques [Kum02]. In this sense, parallel programming research investigates means to approach specific problems through structured parallel programming and means to provide high-level abstractions that lower the effort to apply parallelism in serial algorithms [Kum02, KH10]. Nonetheless, parallel programming research has not yet entirely resolved this problem [MSM04, MRR12, AGDF20].

In 2016, following the parallel programming research line of high-level abstractions for parallel computing, the SPar language was created by Dalvan Griebler in his doctoral thesis [Gri16]. SPar is a Domain Specific Language (DSL) embedded in C++ that offers high-level abstractions for stream parallelism through code annotations with C++ attributes [Gri16, GDTF17]. The SPar compiler makes source-to-source transformations replacing the C++ annotations with parallel code. Figure 1.3 shows the current state of the research about SPar. The sections where this Master's Thesis intends to contribute are marked using a dashed line.

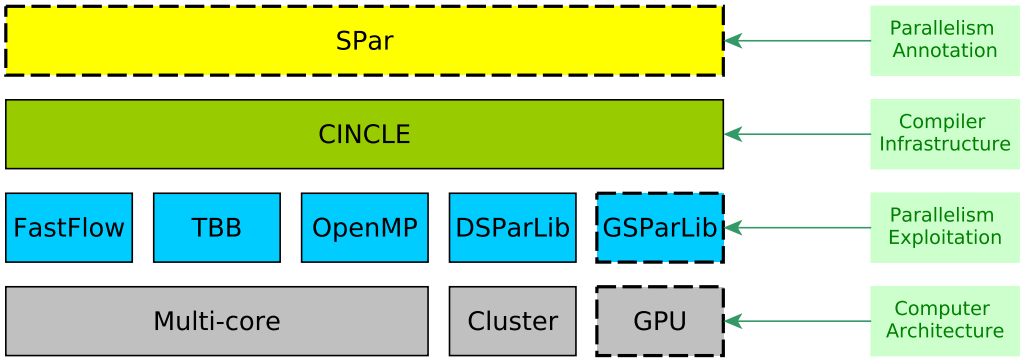


Figure 1.3 – Current state of SPar's Research.

The initial version of SPar supported parallelism on multi-core architectures generating calls to the FastFlow [ADKT17] library. However, more recent studies enabled SPar to generate TBB [HGDF20] and OpenMP [HLG22] code for multi-core architectures, DSParLib (which uses MPI as backend) code for clusters [Pie20], and GSParLib (which uses CUDA and OpenCL as backends) for GPUs [Roc20]. Recent work added support for data parallelism exploitation in the CPU [LHGF21], while the SPar original version gen-

erates only stream parallel patterns for the CPU. At the same time, recent research introduced a self-adaptive degree of parallelism in the CPU [VGF21]; It allows SPar to dynamically change the number of CPU threads to boost its performance during the execution of a stream application. Other studies have applied the parallelism of SPar in several applications compared to lower-level APIs for multi-cores like POSIX Threads [GHDF18a] or studying optimizations for stream processing such as different ways to sort stream data [GHDF18b]. Some works combine the use of SPar with other frameworks such as MPI [VRJ+20] and CUDA [SGDF19]. In summary, the works reported that SPar could perform similarly to manual implementations while requiring significantly lower programming effort [HGDF20, HLG22, Pie20, Roc20, LHGF21, GHDF18a, GHDF18b, VRJ+20, VGF21].

Accelerators such as GPUs are relevant for improving the performance of stream processing applications that offer the opportunity for exploiting data parallelism and are computationally intensive. However, programming applications from the stream domain to efficiently exploit combined parallelism between multi-core CPUs and many-core GPUs or other many-core accelerators is challenging [RSG+19, BFH+04, UGT09, APD+14, APD+15, SRG+20]. Thus, in 2020 Rockenbach extended SPar in his master's thesis targeting GPU programming [Roc20]. The extension proposed on top of SPar offers high-level abstractions for stream parallelism targeting computer architectures composed of CPUs and GPUs [Roc20]. Computer architectures composed of CPUs and GPUs are named heterogeneous computer architectures [NVI20a]. This new implementation allows SPar to combine stream and data parallelism. SPar assigns the stream parallelism to the CPU and the data parallelism to the GPU. The CPU cores control the data flow while the GPU processes each data element with a massive number of threads. The SPar GPU extension uses GSParLib API as the backend to generate GPU code.

GSParLib is a GPU framework based on structured parallel programming that offers abstractions for parallel patterns and wrappers for CUDA and OpenCL. Rockenbach developed GSParLib to be used as a runtime library for other tools, such as SPar, and as a standalone API [Roc20]. GSParLib provides significant benefits compared to the literature. GSParLib aims to support both CUDA and OpenCL GPU backends via a unified way of programming, while most other frameworks support only one. Moreover, GSParLib provides a set of transparent mechanisms to the user; these mechanisms are necessary to implement stream processing applications such as thread safety for manipulating the GPU and asynchronous GPU kernel launches. Additionally, GSParLib also provides abstractions such as batch processing that are relevant optimizations for stream applications. Nonetheless, the support for GPUs on SPar (via GSParLib) is still in an initial stage and can be further improved. This work is motivated by some GSParLib limitations: 1) GSParLib only provides basic features for GPU processing; 2) It only provides two parallel patterns; 3) It was not tested under computationally intensive data parallelism scenarios; 4) It was not tested under scenarios where there is more than one parallel stage in a stream application pipeline; 5) It

has no specific optimizations for GPUs; 6) Its programmability was not evaluated concerning the ease of use and flexibility to express data and stream parallelism on robust applications.

When programming applications targeting accelerators such as GPUs, different problems must be approached differently to achieve relevant speedups [KH10, NVI20a]. For this purpose, a GPU framework must provide a methodology to access the GPUs resources and mechanisms. At the same time, in terms of structured parallel programming, several parallel patterns can be supported. In our previous study [AGDF20, AGR+21], we implemented the NAS Parallel Benchmarks (NPB) [BBB+94] with CUDA and reported our experience implementing it. As an example of how different parallel patterns are essential, we highlight the CG benchmark (part of NPB). CG is characterized mainly by irregular computations that require isolation to achieve a relevant performance on GPUs. Suppose we use a traditional `map` parallel pattern to apply the GPU parallelism. In that case, the GPU achieves a very low speedup (3× the speedup over the serial code [AGDF20]). However, when we use a variation of the `map` parallel pattern that enables organizing subsets of data as tasks, we can isolate irregular computations before offloading them to the GPU. It eliminates divergent instructions and memory misses between the GPU threads, providing higher speedups (70× the speedup over the serial code [AGDF20]).

Another NPB study case is the FT benchmark. FT routines are composed of a complex instruction flow. Upon refactoring the routines and splitting them into data stages where the instruction flow is simplified, the computations become more suitable for the GPU threads. This approach can potentially double the performance of the GPU compared to the traditional approach using the `map` parallel pattern [AGDF20]. Both study cases were implemented manually with CUDA and represent a challenging task for programmers. They are examples of functionalities that could be abstracted and added to GSParLib or SPar to improve the programmability and performance. Additionally, as reported in the literature [KH10, Coo13, CGM14, SK10, NVI20a], specific optimizations for GPUs are crucial for achieving good speedups in these architectures. Among others, GPU-specific optimizations can include memory coalescing, efficiently exploiting the GPU memory hierarchy, and lowering the number of branch divergences [KH10, NVI20a, AGDF20, DKO+19]. The availability of a flexible set of parallel patterns and abstractions to access GPU resources is crucial to guarantee a relevant performance in a framework for GPUs.

Therefore, the main question that drives this research is: **can GSParLib and SPar offer an interface flexible enough to approach robust applications with data and stream parallelism while presenting a comparable performance with state-of-the-art GPU frameworks?** In order to answer this research question, we provide the following contributions:

- **A methodology to provide a unified interface for parallel programming frameworks.** We present a method to develop more agnostic frameworks independent of low-level programming interfaces. The method mainly involves the manipulation of

Strings to manage and generate code of different parallel programming interfaces. Although we applied it to provide a unified interface for CUDA and OpenCL in GSParLib, those techniques can be used in other areas to improve code portability or provide programming abstractions. Our results demonstrated a negligible performance penalty when using this methodology.

- **A set of data and stream parallelism optimizations for GSParLib.** We provide new mechanisms for the limitations found in GSParLib, including, among others, code generation for CUDA and OpenCL syntax, parameters to specify the reuse of data to decrease communication between the CPU and GPU, and abstractions for atomic operations. The modified version of GSParLib can achieve a performance equivalent, or very similar, to handwritten CUDA programs on robust applications such as those present in NPB. Additionally, the improvements consolidate a unified interface for CUDA and OpenCL in the GSParLib, which lowers the programming effort and allows code portability between GPUs of different vendors.

- **A discussion about possible improvements to SPar.** We describe a set of possible optimizations related to the limitations found in SPar and discuss how the SPar compiler can integrate these optimizations. In order to validate the performance improvement of such optimizations, we perform semi-automatic code generation with the SPar compiler and the modified version of GSParLib. When using the optimized code, SPar's GPU extension achieves results similar to manual implementations using lower-level APIs for GPU programming.

- **A robust stream processing benchmark that approaches stream and data parallelism.** A limitation present in the stream processing domain is the lack of available benchmarks. There are even fewer options for evaluating GPUs. We provide a new synthetic benchmark for the stream processing domain. Military Server Benchmark (MS, as we describe in Section 3.3) is a stream processing benchmark that offers the opportunity to combine stream and data parallelism. The benchmark is computationally intensive for GPUs and can be used for testing different GPU programming techniques. MS is also highly customizable. It offers an integrated data-set generator and integrated correctness verification.

- **A performance and programmability evaluation on CFD domain applications using structured parallel programming for GPUs.** We provide an implementation of the NPB kernel benchmarks using the GSParLib's Pattern API, which is based on structured parallel programming. NPB is a suite of benchmarks based on the CFD domain. NPB is composed of representative problems of data parallelism and requires challenging parallelism strategies to present relevant speedups on GPUs. Nonetheless, the available GPU frameworks based on structured parallel programming do not explore the NPB suite.

This document is organized as follows: Chapter 2 presents the background for this work. Chapter 3 presents the study of performance and programmability regarding GSParLib. Chapter 2.5 presents the study of performance and programmability regarding SPar. Chapter 5 presents the related work regarding SPar's GPU extension. Chapter 6 presents our conclusions.

2. BACKGROUND

2.1 Graphics Processing Units

GPUs were initially developed to process 2D and 3D graphics to support the demand of the industry of games. Lately, NVIDIA introduced the Computer Unified Device Architecture (CUDA), which permits the programmers to use the GPUs for general-purpose computations with a small set of extensions of the C language. This way, GPUs became a very popular accelerator that allows programmers to solve a significant set of complex problems, such as Deep Learning [NVI20a, KH10].

2.1.1 General Characteristics

Figure 2.1 presents general characteristics of CPU and GPU architectures.

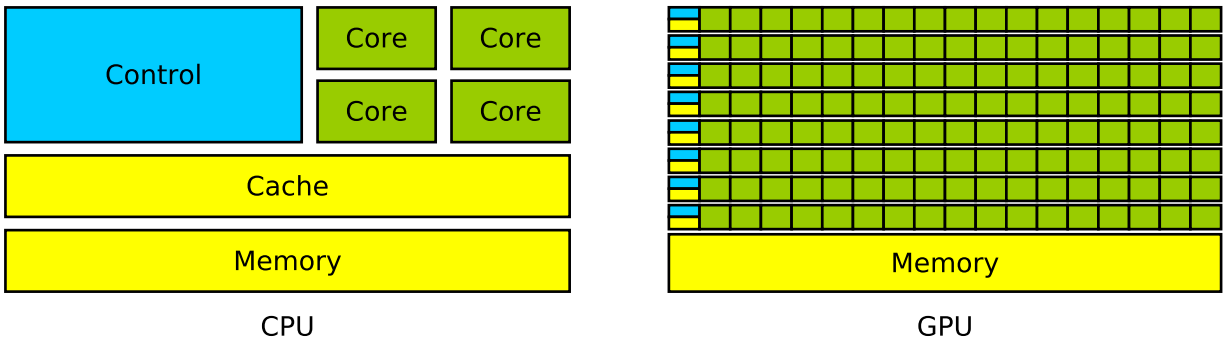


Figure 2.1 – General characteristics of CPU and GPU.

CPUs have a few large and robust cores, a complex control unit, a big amount of cache, and also, the cores have a high clock speed. These characteristics mean that CPUs have a very optimized code for performing complex instruction flows, using cache and mechanisms such as branch prediction [NVI20a, KH10].

GPUs are the opposite of CPUs. GPUs have up to thousands of cores, but the control units are simplified, the amount of cache is small, and the cores have a lower clock speed. This set of characteristics implies that the GPUs are not optimized for running serial code. The GPU threads have low performance on algorithms with complex instruction flows. Complex instruction flows are often characterized by loops, conditional statements, and pipelined instructions. GPUs perform better on algorithms with more straightforward routines and intensive arithmetic operations, which are the main characteristics of computer

graphics algorithms. The standard way to use GPUs is to run part of the program on the CPU and offload intensive parts to the GPU [NVI20a, KH10].

2.1.2 Thread Hierarchy

Current high-end GPUs have up to thousands of cores. They can run millions of threads [NVI20a, SK10, NVI20b], which are organized hierarchically. A function offloaded to the GPU is called kernel and is executed by a grid. A grid contains blocks, and each block contains threads. Blocks and threads can be organized in one, two, or three dimensions, and the number of blocks and threads are also configurable. Each thread identifies its position in the block, and each block identifies its position in the execution grid. Thus, we calculate the global id of a thread by collecting its position in the thread block and thread block position in the grid [NVI20a, KH10].

Figure 2.2 presents an example of a grid of threads. In the example, the grid has four blocks organized in two dimensions. Each block has twelve threads also organized in two dimensions.

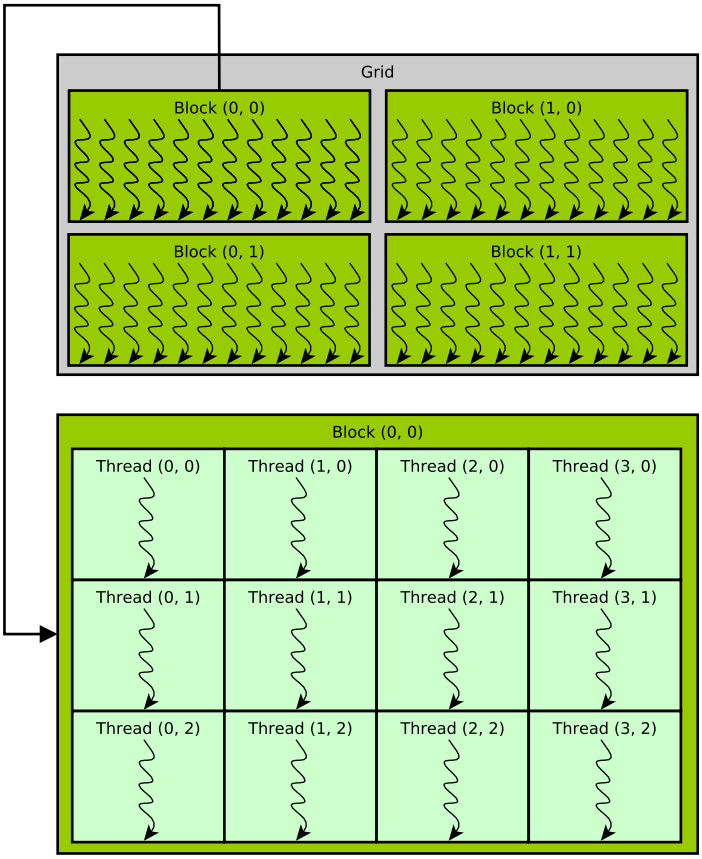


Figure 2.2 – Example of a grid of GPU threads.

2.1.3 Memory Hierarchy

GPUs have their own memories, allowing them to get faster access to data. If the CPU and GPU were to use the same memory, GPU accesses would be slower, and the CPU and GPU would compete for accesses in the memory, reducing overall performance. Thus, when software is written for heterogeneous systems containing GPUs, it is necessary to copy the data from the host to the GPU and perform the computations over the data in the GPU. Then the results are copied back from the GPU memory to the host memory [NVI20a].

The GPU memory hierarchy is divided into localization and visibility. Figure 2.3 presents the physical localization of GPU memory. *Off Chip Memory* has high access latency while *On Chip Memory* has low access latency.

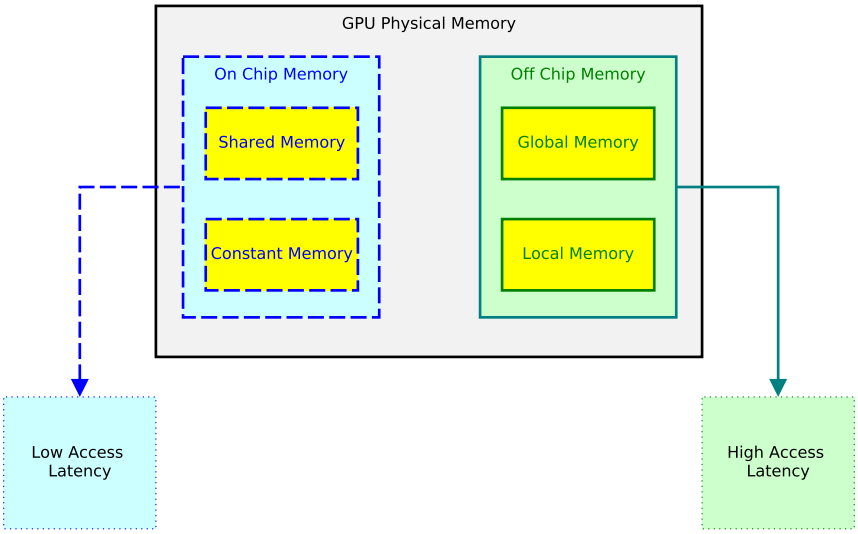


Figure 2.3 – Physical localization of GPU memory.

Figure 2.4 presents the visibility of GPU memory. *Local Memory* is private to a thread. *Shared Memory* is visible to a block of threads. *Global Memory* and *Constant Memory* are visible to any thread of any grid.

Global Memory is the standard way to access GPU data and has a high access latency. Although *Local Memory* is also localized off the chip, their accesses are optimized compared to the *Global Memory* because accesses to the *Local Memory* are automatically coalesced by the GPU [NVI20a, KH10] (we explain memory coalescing in Section 2.1.5). *Shared Memory* has low access latency and works as a cache memory managed by the programmer, as it is only visible to a thread block. It is commonly used in cooperative computations between the threads of a block or to reduce the access latency. *Constant Memory* is a read-only memory. It is used for data that is frequently read but is never written [NVI20a, KH10].

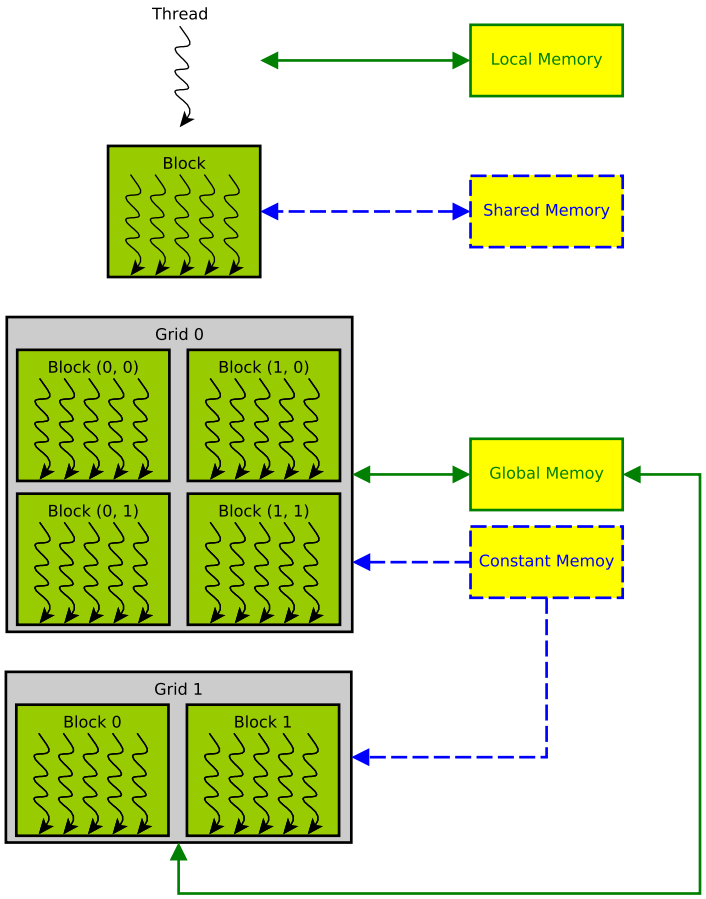


Figure 2.4 – Visibility of GPU memory.

2.1.4 Execution Model

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

GPU stream multiprocessors are responsible for executing thread blocks. When a grid of thread blocks is sent to the GPU, the GPU distributes the thread blocks to the SMs. Then the SMs execute the thread blocks in parallel. The more blocks are created, the more blocks can be executed in parallel. However, since each SM has a limited number of registers, if there are not enough registers, the SM will execute fewer thread blocks in parallel [NVI20a, KH10, SK10, Coo13, CGM14].

Figure 2.5 illustrates how GPU stream multiprocessors work. In the example, a grid with four blocks is sent to a GPU that contains two SMs. The GPU distributes two blocks to the `SM 0` and two blocks to the `SM 1`. `SM 0` executes the blocks `0` and `2`, and `SM 1` executes the blocks `1` and `3`.

A warp is a small group of threads. The GPU split blocks of threads in warps to execute them. The model execution of warps is called Single Instruction Multiple Threads

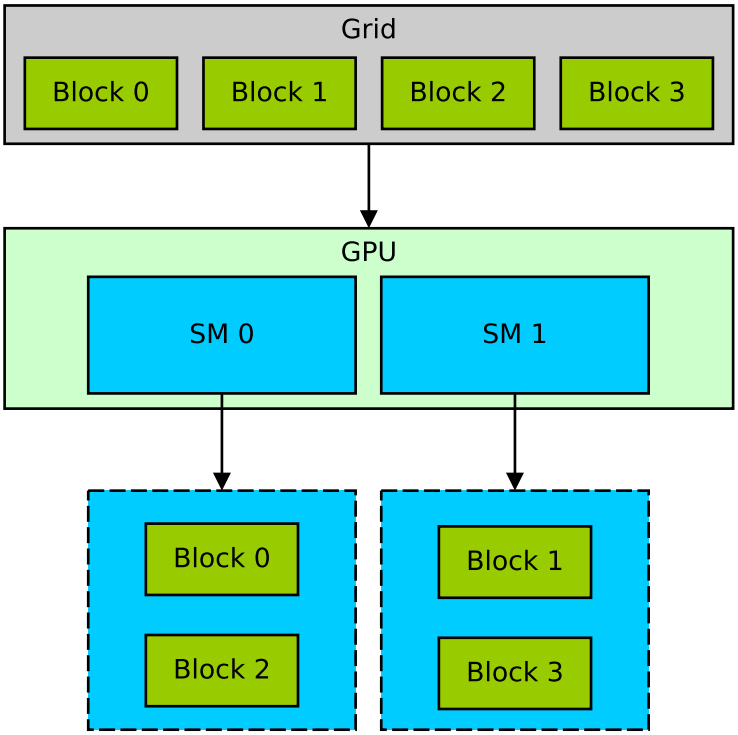


Figure 2.5 – Schedule of thread blocks in the GPU SMs.

(SIMT). The GPU executes only a common instruction between the threads of a warp at a clock cycle. When threads have different instructions, as occurs when the code has loops or conditional statements (occasioning branch divergences), threads of divergent instructions stay inactive. The warp only returns to full execution when all threads have the same instruction again. The more complex the instruction flow, the more the GPU threads' performance penalty. The simplified control unit is one of the reasons why more cores are allowed in GPUs [NVI20a, KH10, SK10, Coo13, CGM14].

Figure 2.6 presents an example of a warp execution. The threads execute together the instructions `I.0` and `I.1`. Since `I.1` is a conditional statement `if/else`, there is a branch divergence with two sets of instructions. The instructions `I.2` and `I.3` are executed only by the threads that entered `Path 0` (`T.0`, `T.1`, `T.2` and `T.3`), while the threads `T.4`, `T.5`, `T.6` and `T.7` stay inactive. When the `Path 0` execution finishes, the `Path 1` is then executed by the threads `T.4`, `T.5`, `T.6` and `T.7`, and the other threads stay inactive (`T.0`, `T.1`, `T.2` and `T.3`). The full warp is only executed again in the instruction `I.6` that is common between all threads.

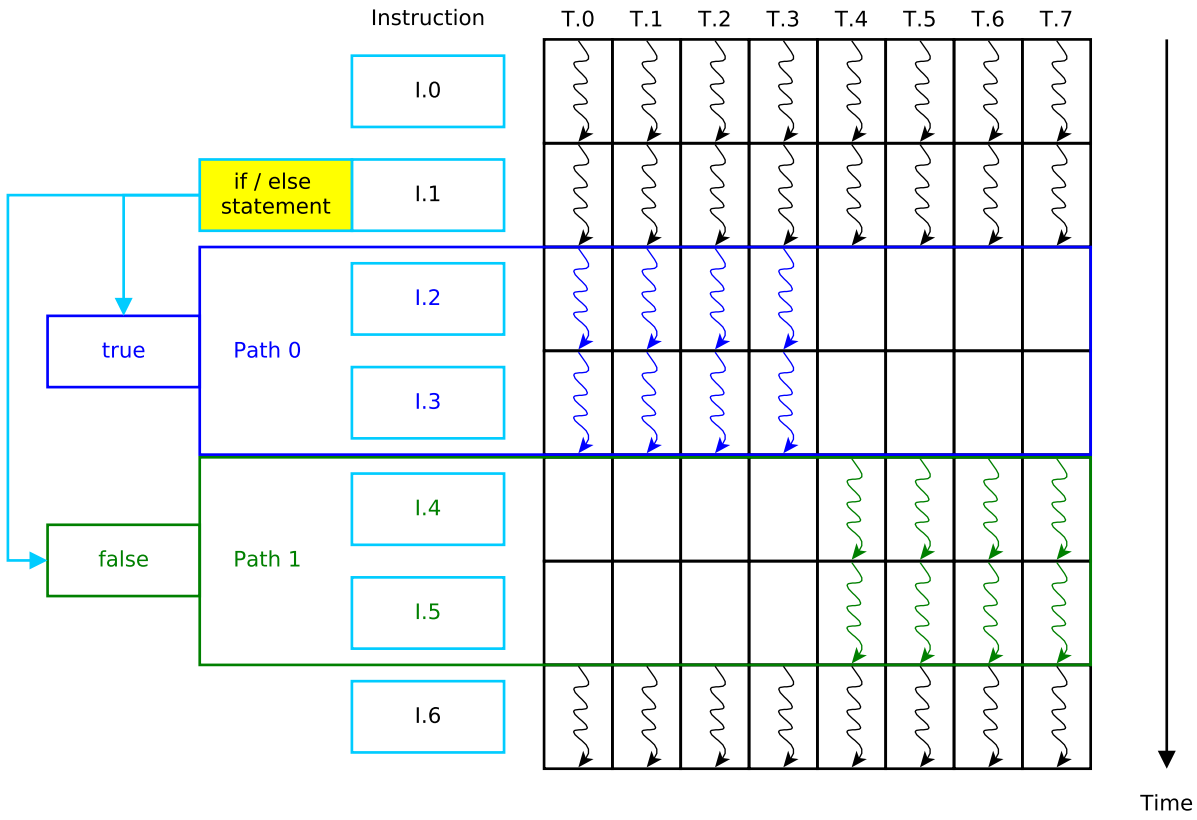


Figure 2.6 – Execution of a thread warp.

2.1.5 Memory Coalescing

GPUs' memory operations are costly and take several clock cycles to execute. As explained in Section 2.1.4, the GPU execution model is SIMT, and threads in the same warp always execute a common instruction in a clock cycle. Accessing the memory is an inherently expensive operation. Thus, it is crucial to optimize these operations. GPUs have an optimization targeting these operations that are called memory coalescing. When GPU threads access contiguous positions in the memory, the GPU hardware automatically makes only a single thread of the warp request the memory operation (instead of each thread executing the operation). Then, the operation is applied in the whole block of memory accessed by the threads (instead of a single position) [NVI20a, KH10].

Figure 2.7 presents an example of uncoalesced accesses. The example illustrates a group of threads executing four iterations of a loop and accessing different positions of an array. At each iteration, each thread walks on different regions of memory. Therefore, the GPU cannot automatically group the operations. Consequently, the GPU performs memory operations sixteen times.

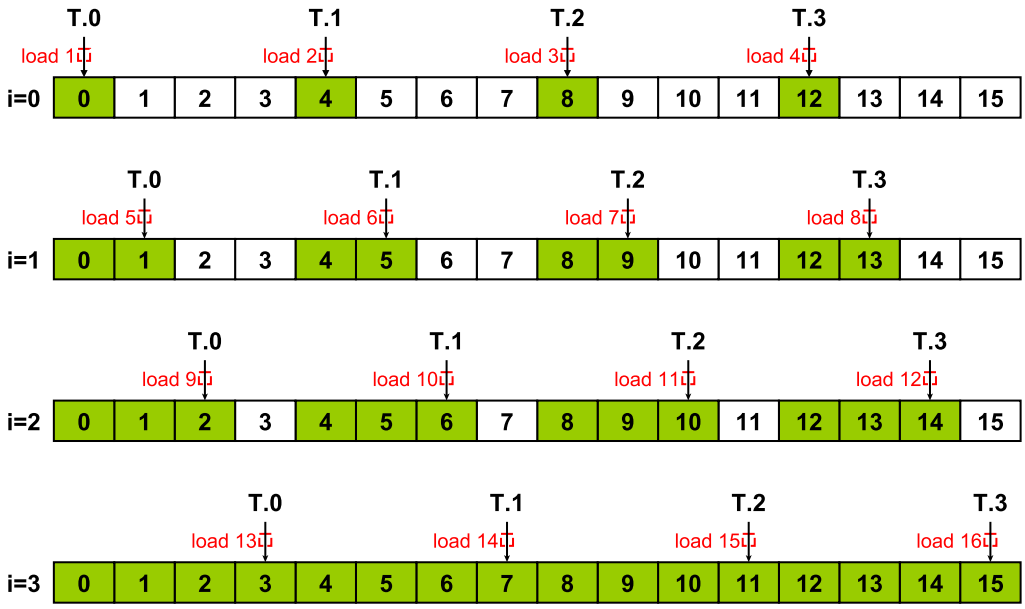


Figure 2.7 – Uncoalesced access pattern.

Figure 2.8 presents an example of coalesced accesses. In this example, the GPU threads access contiguous positions in the memory. Then, at each iteration, the memory instruction `load` is requested by a single thread and loads the entire block of memory for the threads. Consequently, the GPU performs only four memory operations. It is four times fewer than the uncoalesced example.

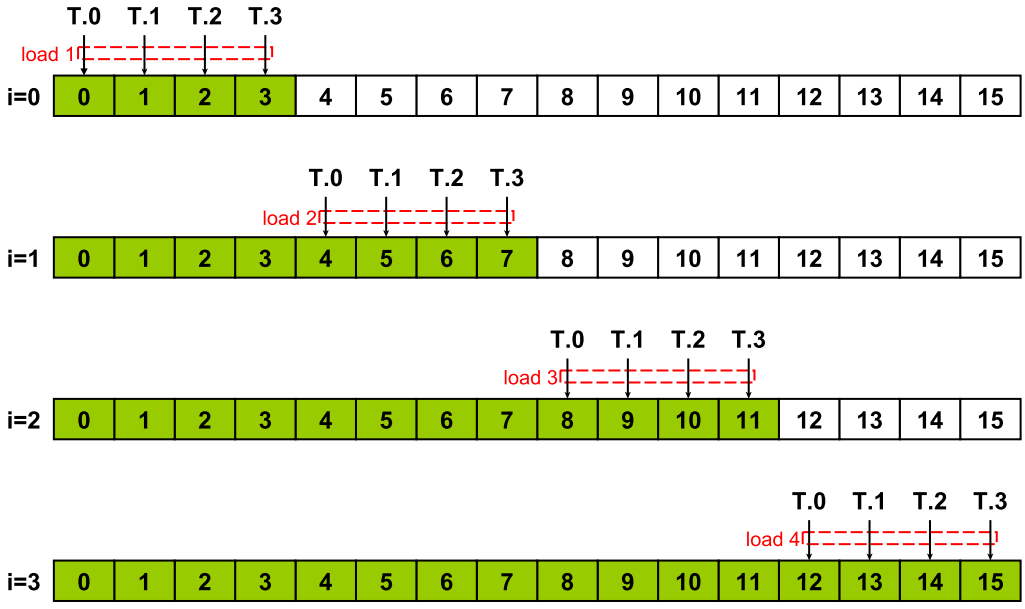


Figure 2.8 – Coalesced access pattern.

2.1.6 GPU Occupancy

Another important GPU concept is occupancy. GPU occupancy is related to using the maximum parallel capacity of the GPU. A way to explore the maximum GPU capacity is to apply fine-grained parallelism, which often requires refactoring the serial code. Avoiding branch divergences also helps the threads to stay active, as explained in Section 2.1.4. Launching several blocks of threads also increases the use of the GPU. Finally, it is possible to execute concurrent GPU kernels [NVI20a, KH10].

GPU occupancy means performance improvement due to the mechanism of thread scheduling performed by the GPU warp scheduler. When a thread requests to execute an instruction that takes several clock cycles to finish, another thread that is ready to run is scheduled. Thus, a GPU core does not stay idle. The literature calls this effect as hiding threads latency and hiding memory latency [NVI20a, KH10, SK10, Coo13, CGM14]. Figure 2.9 illustrates the concept of GPU occupancy.

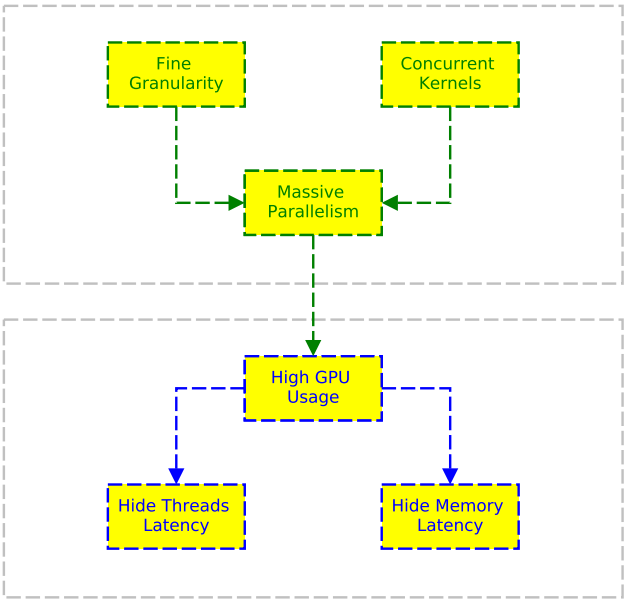


Figure 2.9 – GPU occupancy.

2.1.7 CUDA

CUDA is a framework for GPU programming developed by the manufacturer NVIDIA, suited for their chips. The framework consists of a small set of extensions for the C language. This section shows how to implement an algorithm with CUDA by using a simple matrix multiplication application as an example.

Code 2.1 presents the serial code for a simple matrix multiplication application. The two outermost loops can be executed in parallel in the serial matrix multiplication. Then we create a thread for each combination of iterations from both loops. Each thread corresponds to a single position of the matrices and executes only the innermost loop from the original matrix multiplication algorithm.

```
1 void matrix_multiplication(){
2   for(int i=0; i<N; i++){
3     for(int j=0; j<N; j++){
4       for(int k=0; k<N; k++){
5         matrix3[i][j] += (matrix1[i][k] * matrix2[k][j]);
6       }
7     }
8   }
9 }
```

Code 2.1 – Serial matrix multiplication.

Code 2.2 presents a CUDA implementation for the matrix multiplication algorithm from Code 2.1.

In the CUDA example, in lines `5-7`, we allocate the GPU memory for storing the matrices. In lines `10-12`, we copy the matrices 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 position of the matrices.

In lines `15-18`, we define the thread hierarchy for the GPU kernel. We configure each thread block dimensions as 32 threads on the `x-axis` and 32 threads on the `y-axis`, totaling 1024 threads per block. We define the number of thread blocks in the grid as the number of elements in the matrices divided by the number of threads per block. This definition creates a GPU thread for each position of the matrices. However, suppose the result of the division is not an integer. In that case, we round up this number to create enough blocks of threads to map all elements of the matrices.

In line `21`, we launch the GPU kernel to execute the matrix multiplication algorithm. In line `24`, we define a barrier that forces the CPU thread to wait for the GPU to finish the matrix multiplication algorithm. In line `27`, we copy the results from the GPU memory to the host memory (also called host memory).

In lines `30-40`, we define the GPU kernel that executes the matrix multiplication operation. In lines `31-32`, we calculate the thread's global id. It uses the block's id and the thread's local id. In this implementation, the global id of the thread corresponds to a single position of the matrices. The thread returns in line `35` if its global id does not correspond to a valid position in the matrices. In lines `37-39`, the matrix multiplication is performed. In line `38`, the indexes of the matrices are computed as linear arrays because CUDA only accepts single-dimensional arrays [NVI20a, KH10].

```
int main(){ ...
  size_t size = N * N * sizeof(int);

  //allocating gpu memory
  cudaMalloc(&matrix1_device, size);
  cudaMalloc(&matrix2_device, size);
  cudaMalloc(&matrix3_device, size);

  //copying data to gpu
  cudaMemcpy(matrix1_device, matrix1_host, size, cudaMemcpyHostToDevice);
  cudaMemcpy(matrix2_device, matrix2_host, size, cudaMemcpyHostToDevice);
  cudaMemcpy(matrix3_device, matrix3_host, size, cudaMemcpyHostToDevice);

    //defining grid configuration
  dim3 threadsPerBlock(32, 32, 1);
  int block_x = ceil(double(N) / double(threadsPerBlock.x));
  int block_y = ceil(double(N) / double(threadsPerBlock.y));
  dim3 blocksPerGrid(block_x, block_y, 1);

  //executes the matrix multiplication on GPU
  matrix_multiplication<<<blocksPerGrid, threadsPerBlock>>>(matrix1_device, matrix2_device,
    matrix3_device, N);

  //wait the end of the computation
  cudaDeviceSynchronize();

  //copy data to cpu
  cudaMemcpy(matrix3_host, matrix3_device, size, cudaMemcpyDeviceToHost); ...
}

__global__ void matrix_multiplication(int* matrix1, int* matrix2, int* matrix3, int N){
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  //return if the index is not valid
  if(i >= N || j >= N){return;}

  for(int k=0; k<N; k++){
    matrix3[i*N+j] += (matrix1[i*N+k] * matrix2[k*N+j]);
  }
}
```

Code 2.2 – CUDA matrix multiplication.

2.1.8 OpenCL

OpenCL is a framework developed by the Kronos Group. OpenCL supports several devices such as CPUs, GPUs from diverse vendors (including NVIDIA), and other accelerators. OpenCL is very similar to CUDA. However, OpenCL is more verbose and requires more lines of code. The main differences between OpenCL and CUDA are in naming conventions

and the programming syntax. For example, a block in CUDA is a group in OpenCL, a thread in CUDA is a work item in OpenCL, and shared memory in CUDA is called local memory in OpenCL. More details about the OpenCL syntax are available in the OpenCL Programming Guide [MGM+11].

Code 2.3 presents an OpenCL implementation for the matrix multiplication equivalent to the CUDA implementation from Code 2.2.

```
int main(int argc, char** argv){ ...
  size = N * N;

  cl_context context = 0;
  cl_command_queue commandQueue = 0;
  cl_program program = 0;
  cl_device_id device = 0;
  cl_kernel kernel = 0;
  cl_mem memObjects[3] = {0, 0, 0};
  cl_event finished = NULL;

  context = CreateContext();
  commandQueue = CreateCommandQueue(context, &device);

    program = clCreateProgramWithSource(context, 1, (const char**)&gpu_kernel_function, NULL, NULL);
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "multiplication", NULL);

  CreateMemObjects(context, memObjects, matrix1, matrix2);

  clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
  clSetKernelArg(kernel, 3, sizeof(int), &N);

  size_t local[2];
  local[0] = (size_t) 32;
  local[1] = (size_t) 32;
  size_t global[2];
  global[0] = (size_t) ceil(double(N) / double(local[0])) * local[0];
  global[1] = (size_t) ceil(double(N) / double(local[1])) * local[1];

  clEnqueueNDRangeKernel(commandQueue, kernel, DIMENSION_BLOCKS, NULL, global, threadsPerBlock, 0,
    NULL, &finished);

  clWaitForEvents(1, &finished);

  clEnqueueReadBuffer(commandQueue, memObjects[2], CL_TRUE, 0, size * sizeof(int), matrix3, 0, NULL,
     NULL); ...
}

cl_context CreateContext(){
  cl_int errNum;
  cl_uint numPlatforms;
  cl_platform_id firstPlatformId;
  cl_context context = NULL;
  errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
  cl_context_properties contextProperties[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)
    firstPlatformId, 0};
  context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU, NULL, NULL, &errNum);
  return context;
```

```
49 }
50
51 cl_command_queue CreateCommandQueue(cl_context context, cl_device_id* device){
52   cl_int errNum;
53   cl_device_id *devices;
54   cl_command_queue commandQueue = NULL;
55   size_t deviceBufferSize = -1;
56   errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &deviceBufferSize);
57   devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];
58   errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, deviceBufferSize, devices, NULL);
59   commandQueue = clCreateCommandQueue(context, devices[0], 0, NULL);
60   *device = devices[0];
61   delete [] devices;
62   return commandQueue;
63 }
64
65 void CreateMemObjects(cl_context context, cl_mem memObjects[3], int* a, int* b){
66   memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int) *
       size, a, NULL);
67   memObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int) *
       size, b, NULL);
68   memObjects[2] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(int) * size, NULL, NULL);
69 }
70
71 const char* gpu_kernel_function = "\n" \
72 "__kernel void matrix_multiplication(__global int* matrix1, __global int* matrix2, __global int*
       matrix3, __global int N){ \n" \
73 "    int i = get_global_id(0); \n" \
74 "    int j = get_global_id(1); \n" \
75 " \n" \
76 "    //return if the index is not valid \n" \
77 "    if(i >= N || j >= N){return;} \n" \
78 " \n" \
79 "    for(int k=0; k<N; k++){ \n" \
80 "        matrix3[i*N+j] += matrix1[i*N+k] * matrix2[k*N+j]; \n" \
81 "    } \n" \
82 "} \n" \
83 "\n";
```

Code 2.3 – OpenCL matrix multiplication.

In the OpenCL example, in lines 4-10, we create variables related to the OpenCL API. In line 12, we call a function to create the OpenCL context, that we define in lines 40-49. In line 13, we call a function to create the OpenCL command queue, that we define in lines 51-63. In lines 15-17, we create and build the OpenCL kerne using the string from lines 71-83 (we can write an OpenCL kernel as a string). In line 19, we call a function that we define at the lines 65-69; the function allocates GPU memory to store the matrices and also copies the data from the host to the GPU. In lines 21-24, we set the arguments for the OpenCL kernel. In lines 26-31, we define the thread hierarchy. `local` is the number of threads per block. `global` is the total amount of threads in the grid (in OpenCL, the programmer does not define the number of blocks per grid as in CUDA). In line 33, we execute the OpenCL kernel. In line 35, we call a barrier that forces the CPU thread to wait until the OpenCL kernel finishes its execution. In line 37, copy the results from the GPU to the host. In lines 71-83, we define the computations of the OpenCL kernel. In lines 73-74, we collect the

thread's global id. Unlike CUDA, in OpenCL, it is unnecessary to use the block's id and thread's local id to calculate the global id. In line 77, the thread returns if its global id does not correspond to a valid position in the matrices. In lines 79-81, the thread performs the matrix multiplication. Similar to CUDA, in line 80, we must compute linear indexes to access the matrices' positions.

2.1.9 OpenACC

Like OpenCL, OpenACC is a framework that can run on several architectures, including NVIDIA GPUs. However OpenACC programming model is based on directives, similar to OpenMP. Since OpenACC is a high-level framework, it does not offer some low-level functionalities available in CUDA and OpenCL. For instance, OpenACC does not allow synchronizing the threads from a thread block [Opea, CJ17]. Although the use of directives improves the programmability, it can present performance loss depending on the application due to the lack of flexibility to efficiently approach different parallelism strategies [AGR+21, HMMT13, KH10].

Code 2.4 presents an OpenACC implementation for the matrix multiplications with a similar parallelism strategy adopted in the CUDA and OpenCL examples.

```
1  void matrix_multiplication(){
2    #pragma acc loop
3    for(int i=0; i<N; i++){
4      #pragma acc loop
5      for(int j=0; j<N; j++){
6        for(int k=0; k<N; k++){
7          matrix3[i*N+j] += (matrix1[i*N+k] * matrix2[k*N+j]);
8        }
9      }
10   }
11 }
```

Code 2.4 – OpenACC matrix multiplication.

In the OpenACC example, we need a lower programming effort than CUDA and OpenCL. We apply the directive `#pragma acc loop` on the two outermost loops of the matrix multiplication algorithm. Then, OpenACC creates a GPU thread for each position of matrices. In line 7 it is still necessary to calculate linear indexes to access the matrices' positions. Additionally, we do not need to specify memory transfers between the CPU and the GPU. OpenACC manages the memory automatically.

2.2 Structured Parallel Programming

Structured parallel programming is known in the literature mainly as design patterns, parallel patterns, and algorithmic skeletons [Gri16, MRR12, MSM04]. It refers to a good solution to a specific problem. It registers the experience of experts in the area so that it can provide knowledge to help other programmers solve similar problems [MSM04]. Design patterns originated in the software engineering community, where they provide solutions for object-oriented programming [MSM04]. Figure 2.10 illustrates an overview of parallel patterns.

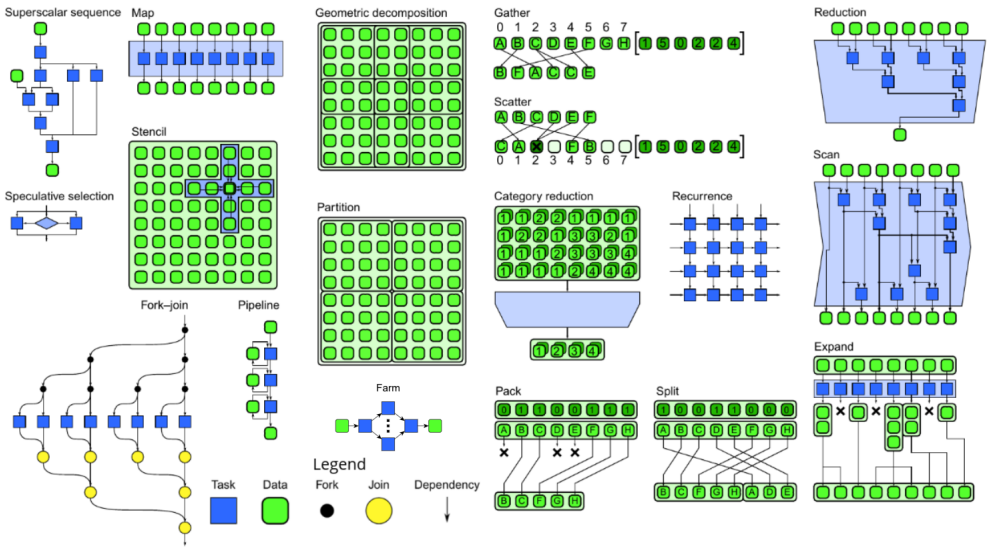


Figure 2.10 – An overview of parallel patterns (original source [MRR12, Gri16, Roc20]).

Based on the literature [Gri16, MRR12, MSM04], we briefly describe each one of the parallel patterns in Figure 2.10 as follows:

- **Fork-join**. A process creates a fork with some other processes to compute other portions of data. A process commonly waits for child processes to terminate its execution. It is useful mainly when the creation of tasks is dynamic during the execution of a program.
- **Map**. Applies the same computation over a set of data defined by an index. It is commonly used when there is a loop where the total of iterations is known, and each iteration is independent.
- **Stencil**. Is a variation of the `map` pattern. It accesses a data element and a set of neighbors. This pattern needs verification of bounds. It is commonly used in imaging processing, for example, operating a computation in a pixel where is necessary info from the neighbors.

- **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. An example of use is computing a summation of a set of integers. We can use the reduction pattern with different operators, such as associative or commutative.
- **Scan**. It is a variation of the `reduction` pattern, it computes every partial `reduction` from a set of elements. We can use this pattern in computations, such as generating random numbers that use a successor function.
- **Recurrence**. It is a generalization of loops where an iteration depends on another one. Recurrence is similar to `map` and `stencil` because a thread or process can use its input and the outputs from neighbors to proceed with the computation. Recurrence can be found in algorithms such as matrix factorization and sequence alignment.
- **Pack**. It eliminates elements that are not being used in a set. Each element is marked with a Boolean. The Boolean indicates if the element is useful or not. The result is a new set of elements maintaining the original order. An example of use for the `pack` pattern is the detection of collision where we want only valid collisions.
- **Split**. `split` is a variation of the `pack` pattern, where instead of removing elements, they are moved to the leftmost or rightmost part of the arrays. This way the `split` pattern does not lose information as the `pack` pattern.
- **Geometric decomposition**. It breaks the data into subsets that can overlap or not. Each subset is assigned to a thread or a process. It can be used in graph and image processing.
- **Partition**. It is a particular case of the `geometric decomposition` pattern where the subsets do not overlap. It can also be applied in the same applications.
- **Gather**. It receives a set of indexes and reads the data only in the specified positions. It is considered a variation of the `map` pattern. Examples of use can be found in sparse matrix operations and collision detection.
- **Scatter**. `scatter` is the inverse of the `gather` pattern, where the pattern writes the data instead of reading it. As the pattern writes data, race conditions can occur if a position is requested at least twice.
- **Superscalar sequences**. This pattern defines a sequence of tasks. The tasks are freely executed concurrently. However, they preserve the order of data dependencies.
- **Speculative section**. In this pattern, both cases of a conditional statement run in parallel. When the conditions finish their execution, the unnecessary branch is canceled, and its side effects are reverted. It is commonly used in compilers to hide the latency of

instructions. To revert a speculative section on CPUs is necessary to cancel the task. This pattern is unsuitable for GPUs because GPU threads only execute a common instruction in the clock cycle.

- **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. Also, the set is ordered according to the order of the threads or processes. This pattern can be used in applications such as compression and decompression.

- **Category reduction**. We can organize this pattern into a sequence of steps: 1) It receives a set of data where each element has a label; 2) It finds all elements with the same label; 3) It applies a `reduction` in each subset using an associative or a commutative operator; Examples of use for the `category reduction` pattern can be found on web analytics computations.

- **Pipeline**. When there is a data flow, and the computation can be decomposed in a sequence of operations, we can implement a `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. The concept is analog to an assembly line. This pattern can be applied in several applications. For instance, when a sequence of filters must be applied to a set of images.

- **Farm**. A `farm` can be seen as a `pipeline` of three stages. The first stage is called the emitter and produces data to be consumed and sent to the second stage. The second stage consists of a set of parallel workers that consume the data sent by the emitter and then send the data to the third stage. The third stage is called the collector. It receives the data processed by the parallel workers from the second stage. It can operate features such as sorting the data before sending it to the output. `farm` is a useful pattern when a `pipeline` stage is a bottleneck. If the stage is stateless, it is possible to process several elements in the stage, improving the performance.

2.3 Stream Processing Applications

Stream processing applications are characterized by a continuous data flow. Stream is becoming very popular as computers are more and more pervasive in modern society. Data for stream applications can come from social media as what people talk about on the internet, from live videos, or several types of sensors. In most cases, a stream is associated with real-time processing. It needs high-performance computing in order to analyze data and produce results of good quality [AGT14]. Figure 2.11 lists some examples of stream processing applications.

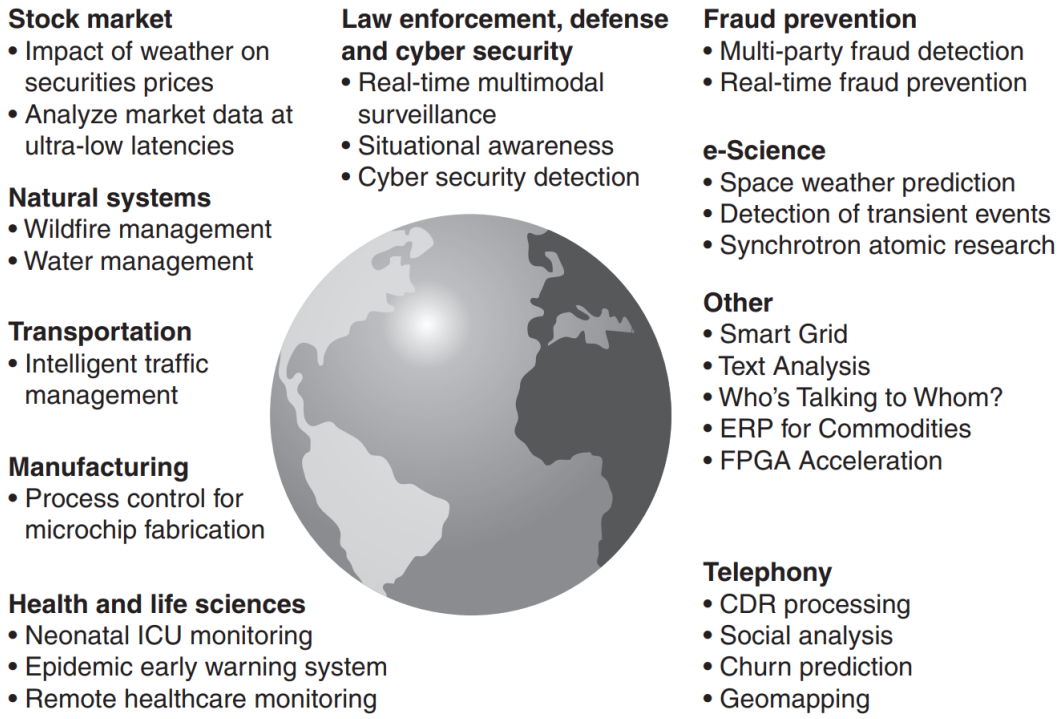


Figure 2.11 – Stream processing applications (original source [AGT14]).

Tuning stream processing applications to be efficient and able to produce results in an acceptable time range is fundamental and requires the use of parallel programming [Gri16]. Using parallel patterns to approach stream processing applications is a way to explore this kind of application [ADKT17] efficiently. The patterns suitable to control the data flow of stream applications are `pipeline` and `farm`. A `pipeline` is used when a stage of a stream is stateful. It means that the data have dependencies. The `farm` pattern can be applied on stages that are stateless, which means that the data have no dependencies. Then, multiple parallel workers can execute the stage by processing different data elements [GDTF17].

A `pipeline` needs a structure to be implemented. It is necessary to implement a queue for communication between each stage. A `farm` implementation also needs a scheduler to assign tasks to the parallel workers. Depending on the structure of communication or schedule, the performance of a stream application can vary. Moreover, some streaming applications require that the application's output follow the input's order. For example, when processing a video, we need to maintain the order of the frames. There are some design alternatives to guarantee the order of the stream elements. For instance, we can implement a sort algorithm in the last stage of the pipeline. Another alternative is creating a linked queue with ordered insertion in the last stage [GHDF18b].

In addition to the patterns that control the flow of data, it is also essential to consider other parallel patterns. For example, `pipeline` and `farm` control the flow of data. However, it is possible to apply other patterns to process each data element of the stream. If a stream application is processing a real-time video, patterns like `map`, `stencil`, or `reduce` can be used

to process each frame of the video. Also, suppose an accelerator as a GPU is present in the hardware. In that case, the GPU can process the frames and considerably improve the performance. This way, exploring a stream application suitably, efficiently, and exploiting all the hardware resources is a complex task [RSG+19].

2.4 GSParLib

GSParLib [Roc20] is an object-oriented C++ library for GPU programming constructed over CUDA and OpenCL. Figure 2.12 presents an overview of GSParLib. GSParLib is divided into two APIs: Driver API and Pattern API; Driver API is a wrapper over CUDA and OpenCL. Pattern API comprises a set of parallel patterns constructed over the Driver API. A standard way to extend GSParLib is by providing new mechanisms in the Driver API and new parallel patterns in the Pattern API.

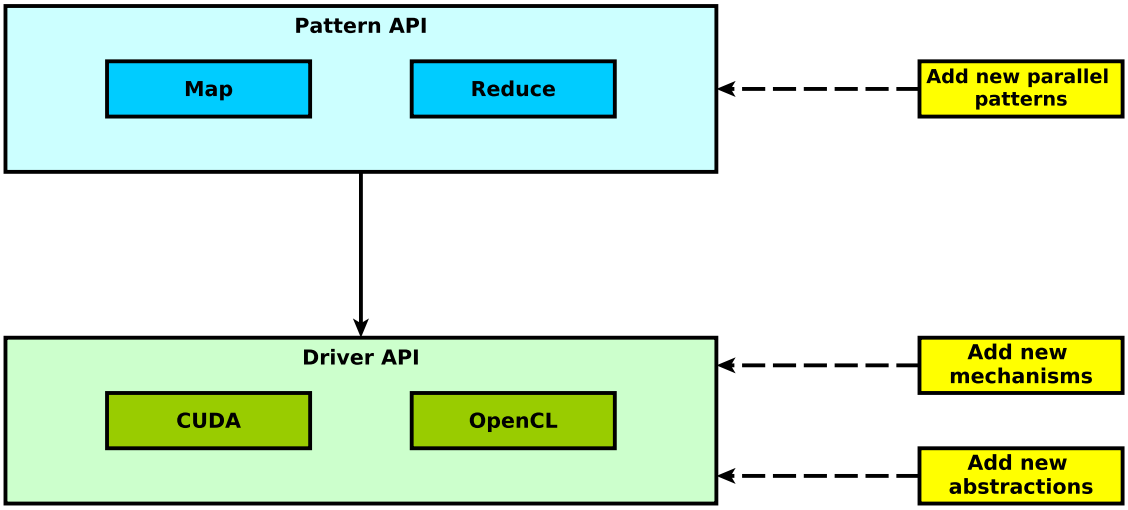


Figure 2.12 – Overview of GSParLib's APIs.

When using GSParLib, we define a GPU kernel as a string and compile it during the execution time. GSParLib follows this strategy for two main reasons: 1) Since the GPU kernel is a string, it easily allows GSParLib to apply transformations in the GPU kernel source; 2) The compilation during execution time allows tuning the applications according to the hardware characteristics collected during the execution time.

The Driver API offers a unified interface for accessing the GPU. It allows functionalities such as allocating GPU memory, copying data between host and GPU memories, and running GPU kernels. In order to avoid racing conditions when manipulating critical GPU resources, such as allocating GPU memory, GSParLib creates a mutually exclusive section for CPU threads. Thus, the programmer can safely exploit simultaneous parallelism using CPU and GPU.

In the GSParLib's original version, we must provide two versions of a GPU kernel, one using CUDA syntax and another using OpenCL syntax, which is a drawback as the programmer must have specialized knowledge in both CUDA and OpenCL.

Code 2.5 shows the matrix multiplication algorithm using GSParLib's Driver API. In lines 1 – 12, we define the GPU kernel with the CUDA syntax. In lines 13 – 24, we define the GPU kernel with the OpenCL syntax. In allows the programmer to switch between CUDA and OpenCL by changing compilation flags. In lines 27 – 38, we initialize the Driver API, allocate the GPU memory, and copy the data from the host memory to the GPU memory. In line 40, we compile the GPU kernel. In lines 42 – 46, we set the parameters of the GPU kernel. In line 48, we define the total number of threads that will execute the GPU kernel. In lines 50 – 56, we run the GPU kernel, wait for it to finish its computations, and copy the memory from the GPU to the CPU.

The Pattern API offers a set of parallel patterns. It conceptually does not require CUDA or OpenCL syntax to write a GPU kernel (which does not work in practice, as shown in Section 3.4). The programmer must identify the stateless code to be executed by GPU threads and use the Pattern API syntax to access the GPU resources. Since the GPU kernel is a string, GSParLib replaces the Pattern API syntax with CUDA or OpenCL calls before compiling it. GSParLib automatically creates CUDA streams and OpenCL command queues when using a parallel pattern, which decreases the programming effort. Those mechanisms are necessary to manipulate simultaneous GPU kernels.

Code 2.6 presents the matrix multiplication application using the Pattern API. In lines 2 – 7, we define the GPU kernel and the `map` pattern instance. In lines 3 – 4, we call functions provided by the Pattern API abstractions. They are replaced by CUDA or OpenCL syntax and collect the thread's id. In lines 8 – 11, we set the parameters of the GPU kernel, setting the amount of GPU memory that must be allocated for each parameter. `GSPAR_PARAM_IN` indicates that the matrices `matrix1` and `matrix2` must copied to the GPU before running the GPU kernel. `GSPAR_PARAM_OUT` indicates that `matrix3` must only be copied from the GPU to the CPU after finishing the computations. In line 12, we compile and run the GPU kernel using $N * N$ threads. In line 13, we delete the `map` pattern, releasing the related resources.

For both Driver and Pattern APIs, upon using the compilation flag `GSPARDRIVER_CUDA`, GSParLib compiles the code with CUDA, and when using `GSPARDRIVER_OPENCL`, GSParLib compiles the code with OpenCL. The Pattern API has three arguments for manipulating memory transfers between the CPU and the GPU. The default argument is `GSPAR_PARAM_IN`, which copies the data from the CPU to the GPU before executing the GPU kernel. `GSPAR_PARAM_OUT` copies the data from the GPU to the CPU after executing the GPU kernel. `GSPAR_PARAM_INOUT` copies the data from the CPU to the GPU before the GPU kernel execution and copies the data from the GPU to the CPU after the GPU kernel execution.

```
#if defined(GSPARDRIVER_CUDA)
// CUDA gpu kernel
#include "GSPar_CUDA.hpp"
using namespace GSPar::Driver::CUDA;
const char* gpu_kernel_source = GSPAR_STRINGIZE_SOURCE(
extern "C"
__global__ void matrix_multiplication(int* matrix1, int* matrix2, int* matrix3, int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    for(int k=0; k<N; k++){
        matrix3[i*N+j] += (matrix1[i*N+k] *
                           matrix2[k*N+j]);}});
#elif defined(GSPARDRIVER_OPENCL)
// OpenCL gpu kernel
#include "GSPar_OpenCL.hpp"
using namespace GSPar::Driver::OpenCL;
const char* gpu_kernel_source = GSPAR_STRINGIZE_SOURCE(
__kernel void matrix_multiplication(__global int* matrix1, __global int* matrix2, __global int*
    matrix3, int N){
    int i = global_id(0);
    int j = global_id(1);
    for(int k=0; k<N; k++){
        matrix3[i*N+j] += (matrix1[i*N+k] *
                           matrix2[k*N+j]);}});
#endif
int main(){ ...
    // driver api initialization
    Instance* driver = Instance::getInstance();
    driver->init();
    auto gpus = driver->getGpuList();
    auto gpu = driver->getGpu(0);
    // gpu memory allocation
    matrix1_device = gpu->malloc(N*N*sizeof(double), matrix1_host);
    matrix2_device = gpu->malloc(N*N*sizeof(double), matrix2_host);
    matrix3_device = gpu->malloc(N*N*sizeof(double), matrix3_host);
    // memory transfer, copy memory to gpu
    matrix1_device->copyIn();
    matrix2_device->copyIn();
    matrix3_device->copyIn();
    // gpu kernel, compiling
    gpu_kernel = new Kernel(gpu, gpu_kernel_source, "matrix_multiplication");
    // gpu kernel, setting parameters
    gpu_kernel->setNumThreadsPerBlockForX(1024);
    gpu_kernel->setParameter(matrix1_device);
    gpu_kernel->setParameter(matrix2_device);
    gpu_kernel->setParameter(matrix3_device);
    gpu_kernel->setParameter(sizeof(int), &N);
    // gpu kernel, setting total amount of threads
    unsigned long dimensions[3] = {N, N, 0}; // N*N threads
    // gpu kernel, running
    gpu_kernel->runAsync(dimensions);
    // gpu kernel, wait the finish of computations
    gpu_kernel->waitAsync();
    // memory transfer, copy memory to cpu (host)
    matrix1_device->copyOut();
    matrix2_device->copyOut();
    matrix3_device->copyOut();}
```

Code 2.5 – Matrix multiplication with the Driver API of GSParLib.

```
void matrix_multiplication(double* matrix1, double* matrix2, double* matrix3, int N){
    Map* map = new Map(GSPAR_STRINGIZE_SOURCE(
      int i = gspar_get_global_id(0);
      int j = gspar_get_global_id(1);
        for(int k=0; k<N; k++){
            matrix3[i*N+j] += (matrix1[i*N+k] *
                               matrix2[k*N+j]);}));
    map->setParameter("matrix1", sizeof(double)*N*N, matrix1, GSPAR_PARAM_IN)
    .setParameter("matrix2", sizeof(double)*N*N, matrix2, GSPAR_PARAM_IN)
    .setParameter("matrix3", sizeof(double)*N*N, matrix3, GSPAR_PARAM_OUT)
    .setParameter("N", sizeof(int), N)
    .run<Instance>({N,N}); // N*N threads
    delete map;
}
int main(){ ...
    matrix_multiplication(matrix1, matrix2, matrix3, N);}
```

Code 2.6 – Matrix multiplication with the Pattern API of GSParLib.

2.5 SPar

SPar is a Domain Specific Language (DSL) embedded in C++ that offers high-level abstractions for stream parallelism through code annotations with C++ attributes [Gri16]. SPar avoids two main burdens of parallel programming: 1) refactoring serial code to explore stream parallelism; 2) writing low-level code to explore the underlying parallel hardware efficiently. These characteristics improve the programmability and maintain a performance similar to a manual implementation with a lower level multi-core framework [GHDF18a,GDTF17].

In order to prevent the programmer from learning a new syntax, SPar uses C++ attributes to express parallelism because the attributes are part of the C++ language. SPar focuses on pipeline and farm parallel patterns, as they are the parallel patterns best suited for stream parallelism. The original version of SPar generates FastFlow [ADKT17] code. Spar uses FastFlow's features and optimizations, such as non-blocking queues, scheduling mechanisms, and sort algorithms for sorting stream data [GDTF17].

SPar Attributes

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 two groups, 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 ID attributes are `ToStream` and `Stage`. `ToStream` identifies the region of the serial code that has the flow of data and should be executed as stream parallelism, commonly a loop as a `for` or a `while`. `Stage` is used to identify each stage of each `ToStream` region.

The AUX attributes are `Input`, `Output`, and `Replicate`. `Input` and `Output` attributes are fundamental to define what data is sent or received by the stages. `Input` identifies the stream's input data (when used with `ToStream`) or the stage (when used with `Stage`). `Output` identifies the output data from the stream or stage. `Replicate` can only be used next to the `Stage` attribute. `Replicate` defines the degree of parallelism of a stage, that is, how many replicas of the stage are executed simultaneously. If the number of replicas is not defined, SPar uses the environment variable `SPAR_NUM_WORKERS`.

The SPar's extension for GPUs added three attributes to SPar [Roc20]. The attributes are `Pure`, `Reduce`, and `Batch`. `Pure` indicates that a loop has no dependencies and can be executed on the GPU. It can be used as an AUX attribute with `Stage` or standalone as an ID attribute. `Reduce` indicates that a reduction operation must be performed. It is an ID attribute. `Batch` expects to receive n data elements before executing the code on the GPU. it must be used as an AUX attribute with the `Stage` attribute. `Batch` decreases communication costs by processing more data at each GPU invocation, increasing the GPU utilization.

SPar Compilation Flags

The SPar compiler accepts three flags that can be passed at compilation time and can modify the execution behavior of SPar. 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 being 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 as communication 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.
- `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.

```
1  [[spar::ToStream]]
2  while(true){
3    element = read_element();
4
5    [[spar::Stage, spar::Input(element), spar::Output(element), spar::Replicate(n)]]{
6      element = computation(element);
7    }
8
9    [[spar::Stage, spar::Input(element), spar::Output(element)]]{
10     write_output(element);
11   }
12 }
```

Code 2.7 – Example of use of SPar.

SPar Compiler

The SPar compiler was built using the infrastructure provided by CINCLE (Compiler Infrastructure for New C/C++ Language Extensions) [Gri16, GDTF17]. CINCLE provides functionalities for C++ code analysis and an API for transformations in ASTs (Abstract Syntax Trees).

Figure 2.13 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 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.

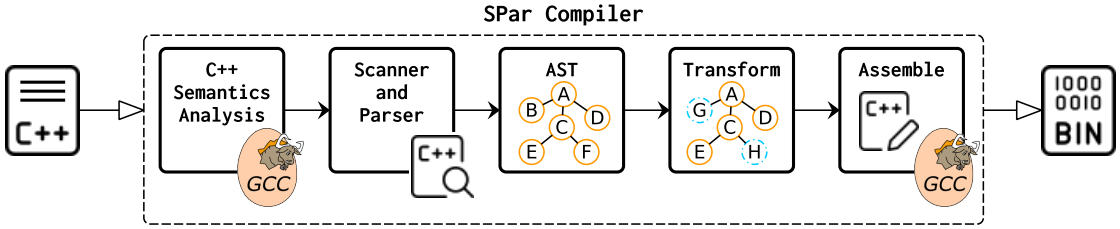


Figure 2.13 – Steps of the compilation process of SPar (original source [Roc20]).

Examples of Code Annotations

This section presents two examples demonstrating how to apply code annotations in a serial code using SPar. Code 2.7 shows an example of using SPar annotations in a multi-core system. In the example, a `while` loop reads, computes, and outputs data elements.

In line `1`, the `ToStream` annotation indicates that the next line (line `2`) loop is a stream region, and the first stage is created. In this example, the first stage reads a data element in line `3`. In line `5`, the attributed `stage` is utilized to indicate the second stage of the algorithm. In this case, the second stage process the element. The attribute `Input` indicates the input

```
1  [[spar::ToStream]]
2  while(true){
3    element = read_element();
4
5    [[spar::Stage, spar::Pure, spar::Batch(size), spar::Input(element), spar::Output(element), spar::
       Replicate(n)]]{
6      for(int i=0; i<element.size; i++){
7        element[i] = pseudo_random_number(i);
8      }
9    }
10
11   [[spar::Stage, spar::Pure, spar::Batch(size), spar::Input(element), spar::Output(element, sum)]]{
12     for(int i=0; i<element.size; i++){
13       [[spar::Reduce]] sum += element[i];
14     }
15   }
16 }
```

Code 2.8 – Example of use of SPar with GPUs.

element of the second stage, and the attribute `output` indicates the output element of the stage. The attribute `Replicate` means that `n` parallel workers will execute the second stage. In line `9`, the third stage is annotated, where the data element is written in the output. In this example, three additional lines of annotations are enough to apply parallelism in the application.

Code 2.8 shows an example of the SPar annotations in a heterogeneous system equipped with a GPU accelerator. In this example, the application receives an array at the first stage, initializes the array with pseudo-random numbers at the second stage, and sums the array's values at the third stage.

Lines `1`, `5`, and `11` identify the data flow using the annotations `ToStream` and `Stage`. In line `5`, the attribute `Pure` is used to indicate that the loop of the line `6` can be executed on the GPU. The attribute `Batch` indicates how many elements must be received before executing the GPU kernel. In line `11`, the attribute `Pure` indicates that the loop of the line `12` can be executed on the GPU. However, in line `13` the attribute `Reduce` is annotated. Then, the SPar compiler will generate a parallel reduction to run on the GPU. The variable `sum` is assigned as the parallel reduction result and stage output.

2.6 Final remarks about the Background

This chapter presented basic concepts of GPU programming, Structured Parallel Programming, and stream processing. Programming GPUs is challenging because it requires deep knowledge of GPUs' hardware, GPU programming techniques, and the target application. The challenge is even more complicated when targeting stream processing applications because applications from this domain additionally require data structures and algorithms to control the data flow. Consequently, the programmer must provide different routines targeting the CPU and GPU and manage their communication. GSParLib and SPar

were developed to facilitate the development of parallel programs, including data parallelism, stream parallelism, and GPUs. However, both GSParLib and SPar are initial approaches.

In the following chapters, we present our study concerning GSParLib (Chapter 3) and SPar (Chapter 2.5). We describe the benchmarks that we implemented to evaluate the programmability and performance of GSParLib and SPar; the limitations that we found in the GSParLib and SPar; the improvements that we provided to GSParLib and SPar; and the experiments that we performed to evaluate the performance of the improved versions of GSParLib and SPar.

3. GSPARLIB EVALUATION AND IMPROVEMENTS

This chapter presents our study of performance and programmability using GSParLib. Sections 3.1, 3.2, and 3.3 describe the benchmarks that we implemented using GSParLib's APIs. Section 3.4 describes the limitations that we found in GSParLib and our optimizations to overcome each one of them. Sections 3.5, 3.6, and 3.7 evaluate the performance and programmability of GSParLib. Section 3.8 describes our final remarks about GSParLib.

3.1 NAS Parallel Benchmarks

This section briefly presents our implementation of the NAS Parallel Benchmarks (NPB) [BBB+94] with GSParLib. This presentation is concise because our latest publications detail our parallelism strategies for those benchmarks [AGDF20, AGR+21, LGM+21].

Although the focus of SPar is stream processing, the GPU is used to apply data parallelism in each stream element. Thus, evaluating specific GPU functionalities of GSParLib requires the evaluation of data parallelism benchmarks besides stream processing ones. Then, to strictly evaluate data parallelism with GSParLib, we implemented parallel versions of the NPB [BBB+94]. For the study, we implemented the benchmarks using both GSParLib's Driver and Pattern APIs.

The NPB [BBB+94] is a consolidated set of benchmarks consisting of five kernels and three pseudo-applications. All eight benchmarks mimic computations extracted from the CFD domain. While the kernels represent the core of CFD applications, the pseudo-applications reproduce entire CFD computations. NPB was initially developed in Fortran and consisted of two versions: a serial version [BBB+94] for reference and an OpenMP version [JFY99] targeting multi-core architectures. Later, other versions for clusters [BHS+95], hybrid programming [WJ03], and grid computing [WF02] were released. The NPB suite provides a set of workloads called classes, ordered by the size as S, W, A, B, C, D, E, and F. Simple tests commonly use S and W classes, while experiments use the other classes. NPB website [NAS] gives a complete description of the workloads. NPB also provides tests of correctness integrated into the benchmarks. They are convenient for validating parallel versions of the benchmarks.

The benchmarks from NPB are primarily composed of iterative procedures (except in the EP benchmark), and the procedures call a set of functions. Figure 3.1 presents a flowchart of the NPB kernels [AGR+21]. Rectangles with solid lines are the functions. Functions offloaded to the GPU are rectangles with a green background. Each benchmark from NPB has an initialization routine and a verification routine. The time measurement

starts after the execution of the initialization routine and finishes before calling the verification routine.

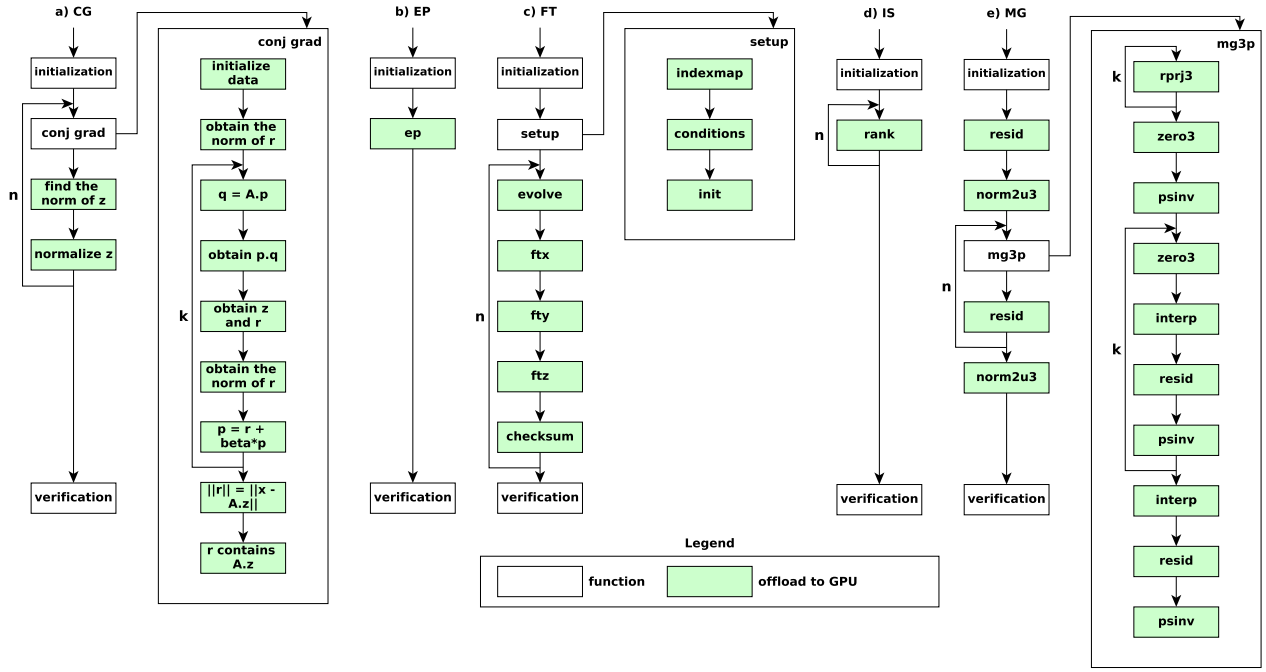


Figure 3.1 – NPB kernels' flowchart (original source [AGR+21]).

We implemented our GPU version using the GSParLib APIs and a C++ conversion from the NPB 3.4 [LGM+21, GLM+18], which strictly follows the original Fortran version. We checked to ensure that our GPU version passed all the NPB correctness tests. In our GPU implementation with GSParLib, we followed the same strategies adopted in our previous works, where we provided a CUDA version for the NPB [AGDF20, AGR+21] (the papers present further details about the parallelism strategies). Since the Driver API is a lower-level interface for parallel programming, the experience of implementing the NPB is similar to CUDA and OpenCL. The implementation with the Pattern API is more affordable than the Driver API because we used parallel programming abstractions via algorithmic skeletons available in the GSParLib. Nonetheless, our parallelization strategies required knowledge and manual insertion of CUDA and OpenCL code even when using the Pattern API, which is a higher-level programming interface. It occurs because GSParLib is still an initial work and does not offer abstractions for all features of those state-of-art frameworks. We cover details about those and other limitations in Section 3.4.

This section briefly described the benchmarks that we used for evaluating data parallelism on GSParLib. The next section presents three legacy benchmarks that we used to evaluate stream processing.

3.2 Legacy Stream Processing Benchmarks

This section briefly describes a set of legacy stream processing benchmarks that we used to evaluate stream parallelism with GSParLib. Rockenbach provided the legacy stream processing benchmarks in his master's thesis [Roc20]. We implemented the parallel versions using FastFlow to control the stream elements' flow and GSParLib's Pattern API to apply data parallelism on each stream element. We describe the benchmarks as follows:

1. **Lane-detection (LD)**. LD is an algorithm used for detecting linear patterns on images. This application is useful mainly for autonomous vehicles. Autonomous vehicles often have an integrated camera that captures several frames per second. The LD algorithm is crucial because it allows correctly planning the trajectory of vehicles [LTA+22]. A robust LD application can be very complex. It should cover variables such as illumination, appearance, and age of a lane marking [LTA+22]. The LD implementation presented by Rockenbach [Roc20] is a simplified version and does not consider such variables.

   Figure 3.2 illustrates the LD benchmark flowchart (the flowchart is the same for the LD, MB, and RT benchmarks). Rounded rectangles are threads. Rounded rectangles with a green background are threads that offload computations to the GPU. Dashed rounded rectangles are stages. Arrows represent the communication between threads of different stages. Circles are stream elements. Three stages compose this LD benchmark. The first stage reads an element from the input. The second stage processes an element by applying a Gaussian and a Sobel filter. The third stage writes an element in the output.

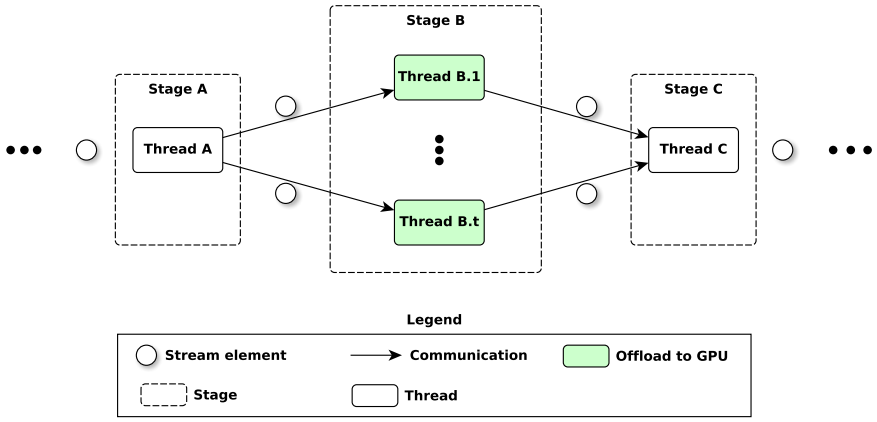


Figure 3.2 – LD, MB, and RT benchmarks' flowcharts.

The LD stream element is a frame captured by a camera. While the CPU controls the flow of frames, the second stage offers the opportunity to employ data parallelism, as it is possible to use the GPU massive parallelism to perform the Gaussian and

Sobel filters. Additionally, the second stage is stateless as it allows the computing of each frame independently. We commonly expect that the more threads are offloading computations to the GPU, the more prominent the GPU utilization will be. Increasing the GPU utilization is beneficial for GPU performance, as presented in Section 2.1.6.

In LD, the computational routines that the CPU offload to the GPU do not require robust strategies or access to GPU resources such as shared memory and atomic operations. In the parallelism strategy, we assign a GPU thread to each position of the frames. Since this task is straightforward, LD requires lower programming efforts than previous NPB parallelizations. Furthermore, LD generates small loads for the GPU and performs many CPU operations and communication between CPU and GPU. In opposite, the NPB allows performing the whole computations on the GPU [AGR+21].

2. **Mandelbrot (MB)**. MB is an algorithm that generates fractal images [YTZ21]. It is defined by the quadratic polynomial $z = z^2 + c$, where c is a constant and z receives the initial value. The polynomial is called several times as an iterative process [ASA21]. Although MB is usually approached as data parallelism when targeting GPUs, Rockenbach [Roc20] adapted it as a stream processing application. In the MB modified version, each row of the generated image is considered a stream element.

   The MB execution flow is equivalent to LD (Figure 3.2). The first stage reads an element. The second stage processes the fractal. The third stage writes an element in the output. The second stage is the one that offers the opportunity for data parallelism. In this stage, we offload each row to the GPU and assign a GPU thread to each row position.

3. **Ray-tracing (RT)**. RT is a method used to simulate the behavior of light in 3D scenes [KSCK19]. This technique is present mainly in applications such as 3D animations and 3D games [KSCK19]. The popularity of those algorithms led NVIDIA to include specialized cores (called RT cores) in their GPUs to compute RT algorithms [WUM+19]. Other fields of study also use RT, such as electromagnetic numerical simulation algorithm [NHJ21] and mesh point location [MWUP20]. The RT benchmark is as simple as LD and MB. The execution flow is equivalent to LD (Figure 3.2). In the second stage, we offload the computations to the GPU and assign a GPU thread to each position of the frames. The first stage reads frames from the input. The third stage sends the frames to the output.

This section briefly described the legacy stream processing benchmarks. The following section presents a robust stream processing benchmark to evaluate better GSParLib.

3.3 Military Server Benchmark

In this section, we present a robust benchmark to complement the evaluation of the GSParLib on the stream processing domain. The legacy stream processing benchmarks introduced in Section 3.2 present limitations for evaluating GPU programming. The main reasons are threefold: 1) they are not compute-intensive for GPUs. 2) they do not fully exploit the underlying GPU resources; 3) they do not require robust parallelism strategies. Thus, we designed a new benchmark application to provide a better and more representative scenario for evaluating GSParLib.

Military Server Benchmark (MS) is a synthetic stream processing application that simulates a military server that continuously receives data collected from drones. The benchmark is composed of heavy computations and allows exploiting data parallelism in each stream element. In the MS simulation, each stream element contains a list of coordinates captured by a drone and a list of military units. The objective of each drone is to find the best coordinate for each military unit from its list. Each military unit has a different type and requisites for proper localization. As analyzing the data is a burdensome task for the embedded system of a drone, the server performs this task.

Figure 3.3 illustrates the MS flowchart. Rounded rectangles are threads. Rounded rectangles with a green background are threads that offload computations to the GPU. Dashed rounded rectangles are stages. Arrows represent the communication between threads of different stages. Circles are stream elements. We can replicate stages B, C, and D because they are stateless.

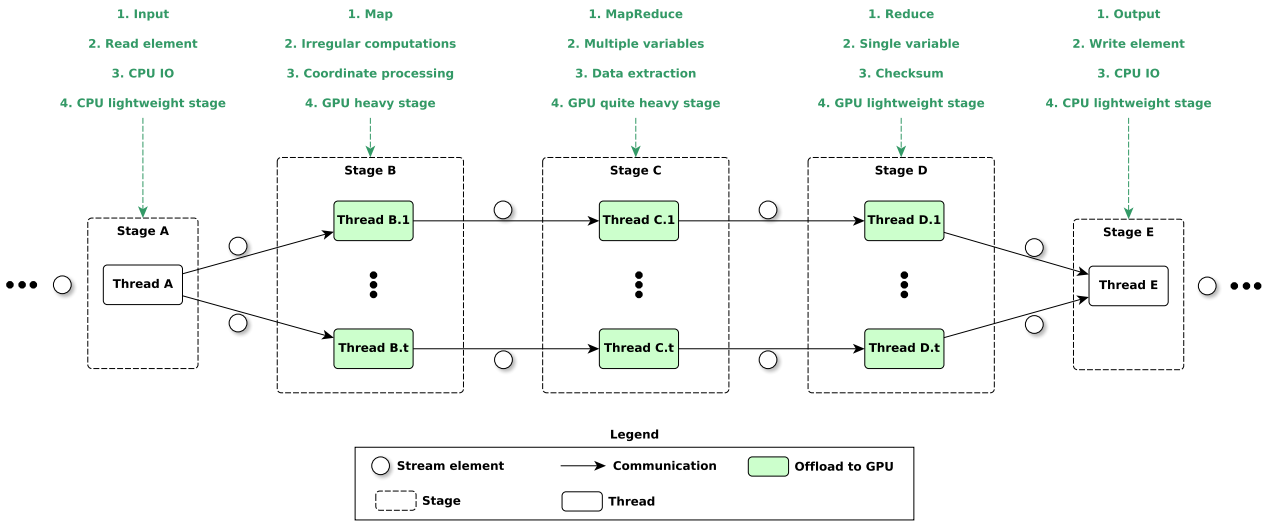


Figure 3.3 – MS Benchmark flowchart.

As in the legacy stream processing benchmarks, we implemented the parallel version of MS using FastFlow for programming the CPU and GSParLib's Pattern API for programming the GPU. We describe the five stages that compose MS as follows:

1. **Stage A**. Stage `A` is a CPU lightweight stage. The main characteristic of this stage is the CPU IO. As it is a synthetic benchmark, the CPU loads the input by reading data from the disk.

2. **Stage B**. Stage `B` is a computational intensive stage performed by the GPU. The main characteristic of this stage is the processing of irregular computations using the parallel pattern `Map`. Stage `B` is responsible for extracting information from each drone's coordinates, such as average height.

   Figure 3.4 illustrates Stage `B`'s flowchart. Rectangles are functions, and rectangles with a green background are functions offloaded to the GPU. When a drone captures data, it selects a $[x, y]$ coordinate and rotates itself four times, generating four data fragments. In this stage, the server iterates over each drone's coordinate and coordinate's rotation, forming a double nested loop. Inside the double nested loop, the server's algorithm executes four procedures: 1) It selects the data fragment from the current rotation. 2) Computes the coordinate's lowest point. 3) Computes the coordinate's highest point. 4) Computes the coordinate's average height. The Big O notation [CLRS09] of Stage `B` is $O(nc * nr * nx * ny)$, where *nc* is the number of coordinates, *nr* is the number of rotations, *nx* is the number of elements on the x-axis of the coordinate, and *ny* is the number of elements on the y-axis of the coordinate.

   Procedure 1 does not perform any loop computation; it just selects a data fragment. In contrast, procedures 2, 3, and 4 perform a double nested loop that iterates over each coordinate's $[x, y]$ position. Moreover, each coordinate has a different size and refers to very different positions in the map, configuring irregular computations. The whole map is also loaded into the memory by the server for inspecting a given drone's coordinates.

   Figure 3.5 gives more details about the procedures from Stage `B` by illustrating how the algorithm computes the average height of a coordinate. Squares with a white background are map positions, and squares with a light yellow background are map positions that are being read by the algorithm. The input is a military unit of type 2, whose reference value is 10.0; the most suitable coordinate for this military unit is the one where the average height is the closest possible to 10.0. Figures 3.5(a), 3.5(b), 3.5(c), 3.5(d) respectively illustrate `rotation 0`, `rotation 1`, `rotation 2`, and `rotation 3` of the computations from Stage `B`. The parameters of this coordinate is $x = 0$, $y = 1$, and $width = 3$ (different coordinates can present different values for x, y, and $width$). The algorithm sequentially extracts the average height of each rotation. The average height from `rotation 0` is 1.0 (Figure 3.5(a)), from `rotation 1`

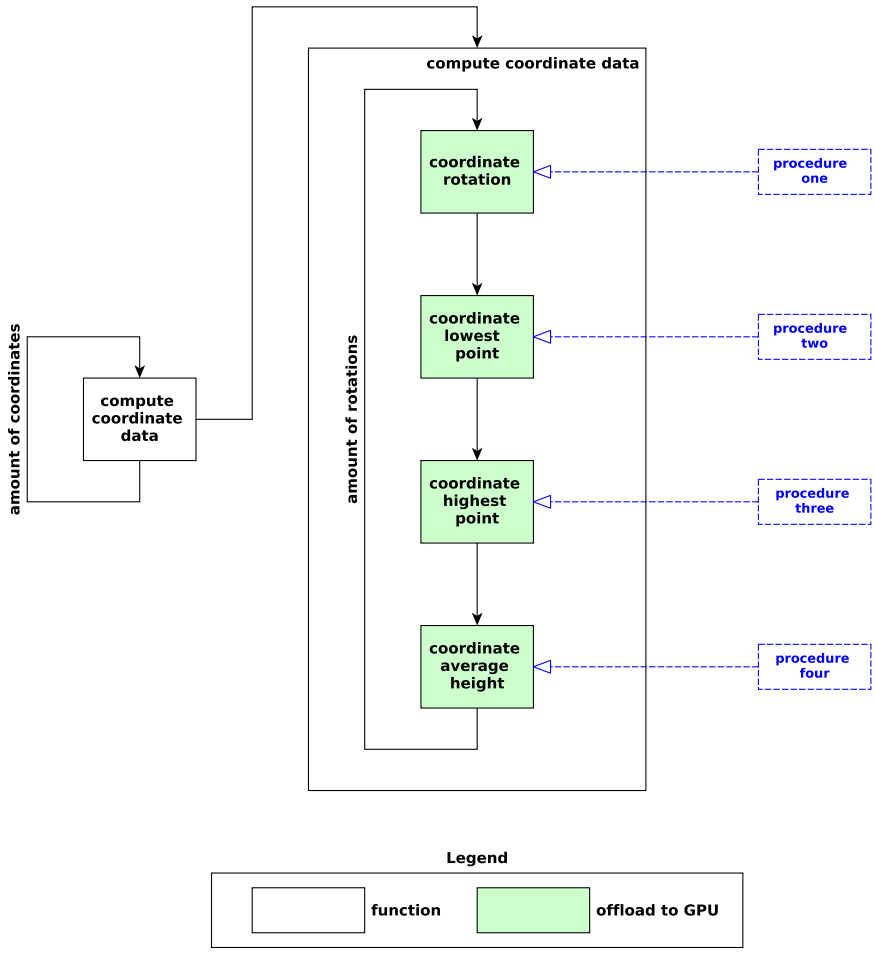


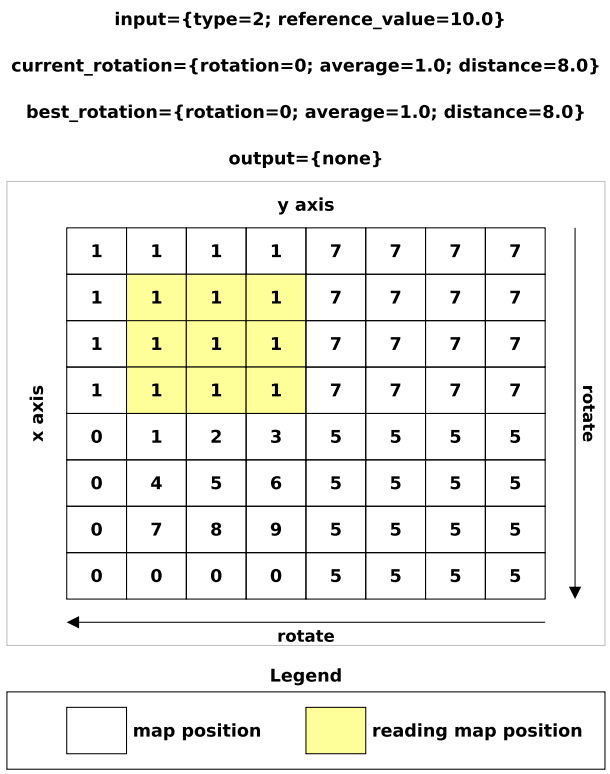
Figure 3.4 – MS Benchmark Stage `B`'s flowchart.

is 9.0 (Figure 3.5(b)), from `rotation 2` is 5.0 (Figure 3.5(c)), and from `rotation 3` is 7.0 (Figure 3.5(d)). As `rotation 1` presents the smallest distance to the reference value of the military unit, the algorithm chooses `rotation 1` as the best option for the military unit. Although this example illustrates the verification of a single coordinate, each drone verifies a list of coordinates to choose the most suitable for each military unit. MS is a flexible benchmark; thus, future work can modify the benchmark to extract other information from maps.

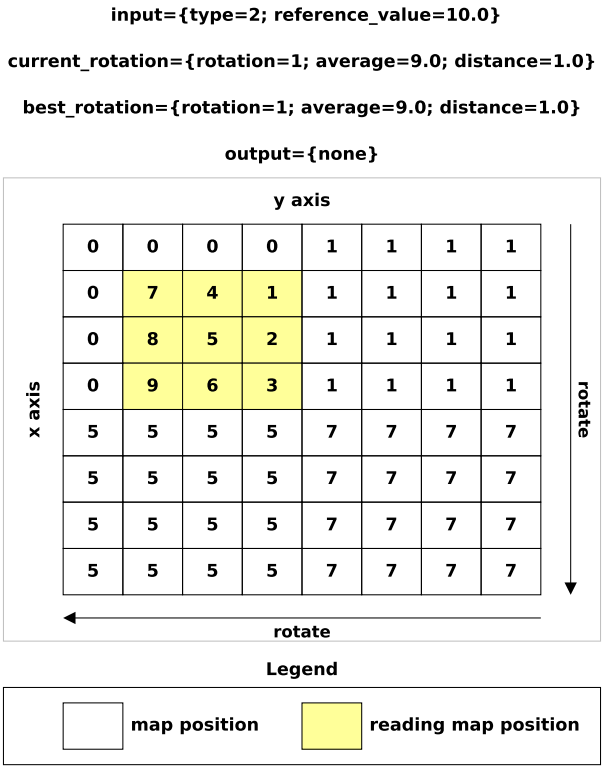
We summarize our parallelism strategy for stage `B` as a sequence of optimization steps:

(a) We merge procedures 1, 2, 3, and 4 into a single routine. The literature names this technique as GPU kernel fusion [KH10]. We perform this optimization primarily using the loop fusion technique. It is possible because the procedures have the same iteration space and no data dependency. GPU kernel fusion is relevant for lowering the amount of GPU kernel launches.

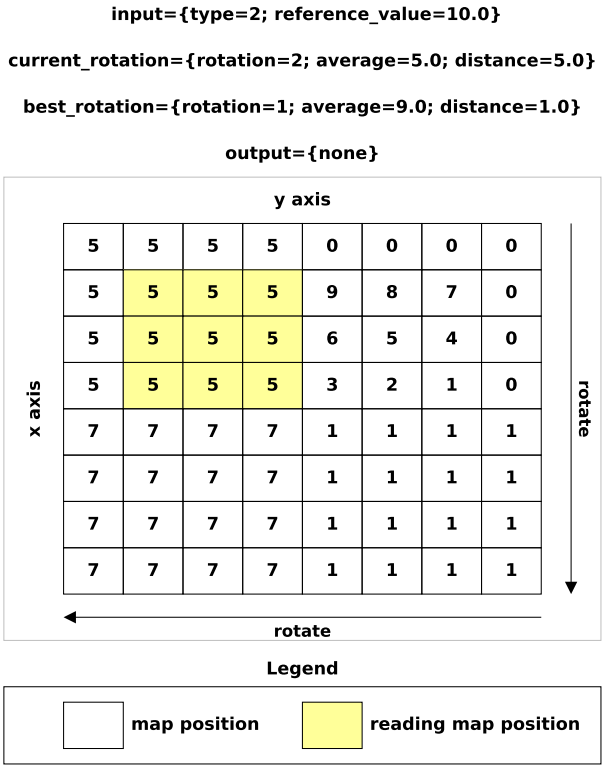
(b) We assign a thread block to each coordinate of a drone to isolate irregular computations, which is critical to allow convergent instructions between the GPU threads and implement coalesced access patterns. Suppose GPU threads from the same



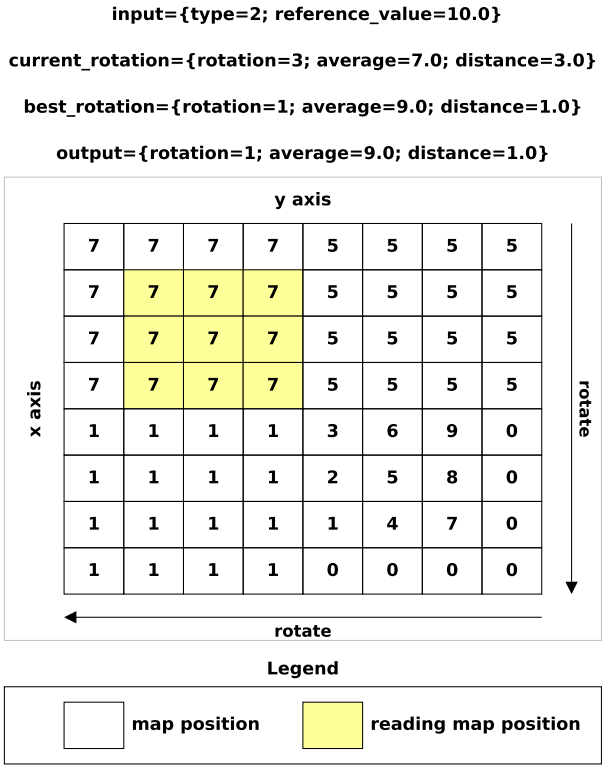
(a) Illustrating `rotation 0` from Stage `B`.



(b) Illustrating `rotation 1` from Stage `B`.



(c) Illustrating `rotation 2` from Stage `B`.



(d) Illustrating `rotation 3` from Stage `B`.

Figure 3.5 – MS Benchmark Stage `B`, computing the best average height of a coordinate for a military unit.

thread block process computations from different coordinates. Each coordinate has a different size and accesses very different map regions. In that case, the instructions between the threads will diverge, imposing a GPU under-utilization because GPU threads execute only a common instruction in a clock cycle. Additionally, as coordinates access different map regions, it is impossible to provide contiguous memory accesses for a group of GPU threads.

(c) We distribute the positions of the coordinate between the threads of the thread block.

(d) We modify the access patterns to allow the GPU optimization of memory coalescing. This optimization is relevant for lowering the memory latency.

(e) We perform most computations using GPU shared memory. GPU shared memory accesses are faster than GPU global and local memory.

(f) We implement a binary tree parallel reduce algorithm for combining results from each thread block [NVI12]. The result of each thread block is the average height and the lowest and highest points of the coordinate. Figure 3.6 illustrates the thread block reduction algorithm. Each thread loads its data on the GPU shared memory in the initialization. Then, the algorithm starts an iterative process. At each step: 1) The algorithm divides in half the number of active threads; 2) A thread combines the results of itself and another thread. 3) The thread block synchronizes its threads; The algorithm stops when only a single thread remains active, which holds the accumulated result.

(g) We apply loop collapsing and unrolling to lower the number of branches in the GPU kernel.

In short, the resulting GPU kernel performs a sequence of two simple loops. In the first loop, the GPU threads compute the coordinate's positions and perform $O(CS/n)$ branches, where CS is the coordinate size, and n is the thread block size. In the second loop, the GPU performs a reduction algorithm in the thread block, and each thread performs $O(log\ n)$ branches, where n is the thread block size. This strategy's total amount of threads is the number of coordinates multiplied by the size of the thread blocks.

3. **Stage C**. Stage `C` is a computational intensive stage performed by the GPU. The main characteristic of this stage is the use of the parallel pattern `MapReduce` with multiple output variables. Stage `C` is responsible for finding the best coordinate for each military unit and uses the extracted data from stage `B`.

   Figure 3.7 illustrates Stage `C`'s flowchart. Rectangles are functions, and rectangles with a green background are functions offloaded to the GPU. Dashed rectangles are routines. This stage is composed of a triple nested loop; the server iterates over each

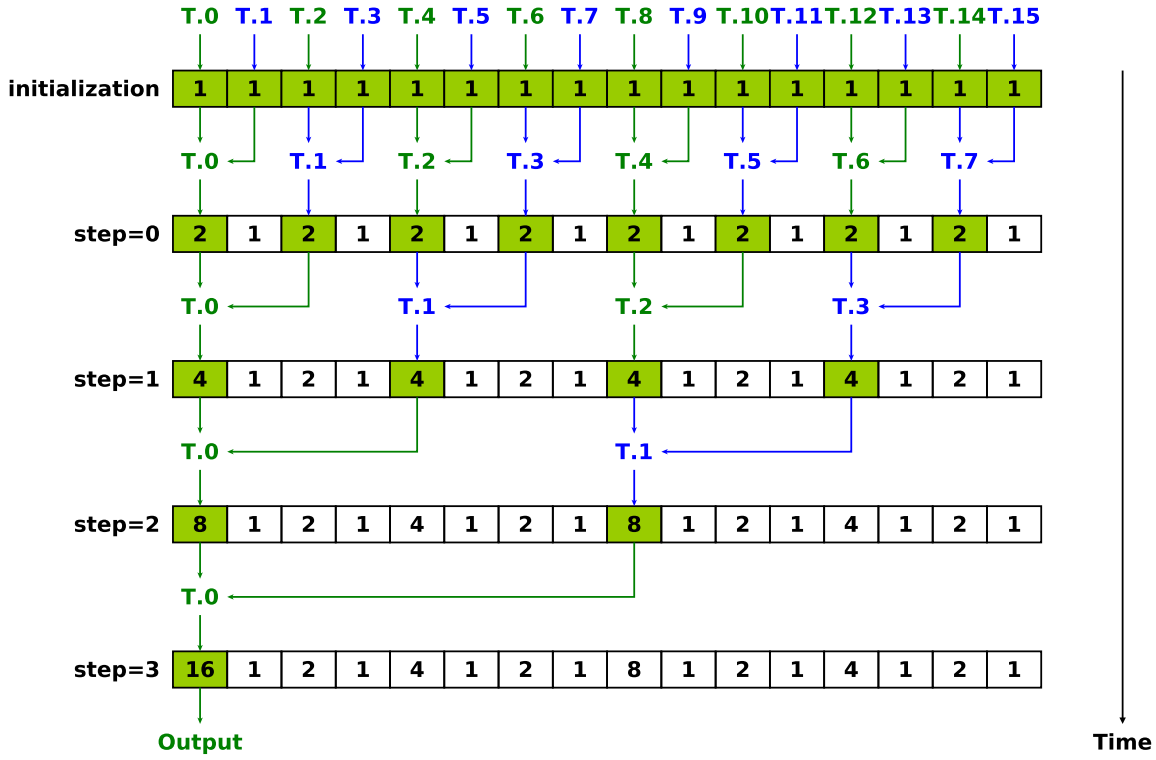


Figure 3.6 – Binary tree parallel reduce algorithm in a thread block.

drone's military unit, each drone's coordinate, and each coordinate's rotation. The stage is composed of two procedures. The algorithm finds the best coordinate for a military unit in the first procedure. In the second procedure, the algorithm updates the utilization of the chosen coordinate for a given military unit. A `MapReduce` pattern is suitable for this problem, where the `Map` pattern extracts the best coordinate for a given military unit. The `Reduce` pattern outputs the number of military units that the algorithm is assigning to this coordinate. This `Reduce` pattern has multiple outputs: 1) The total amount of military units in the coordinate (integer); 2) The total amount of military units per type (array of integers); 3) The coordinate utilization factor (double). The Big O notation [CLRS09] of the stage `C` is $O(nm * nc * nr)$, where nm is the number of military units, nc is the number of coordinates, and nr is the number of rotations.

We summarize our parallelism strategy for stage `C` as a sequence of optimization steps:

(a) We assign a thread block to each military unit.

(b) We distribute the drone's coordinates between the threads of the thread block.

(c) We modify the access patterns to allow the GPU optimization of memory coalescing.

(d) We perform most computations using GPU shared memory.

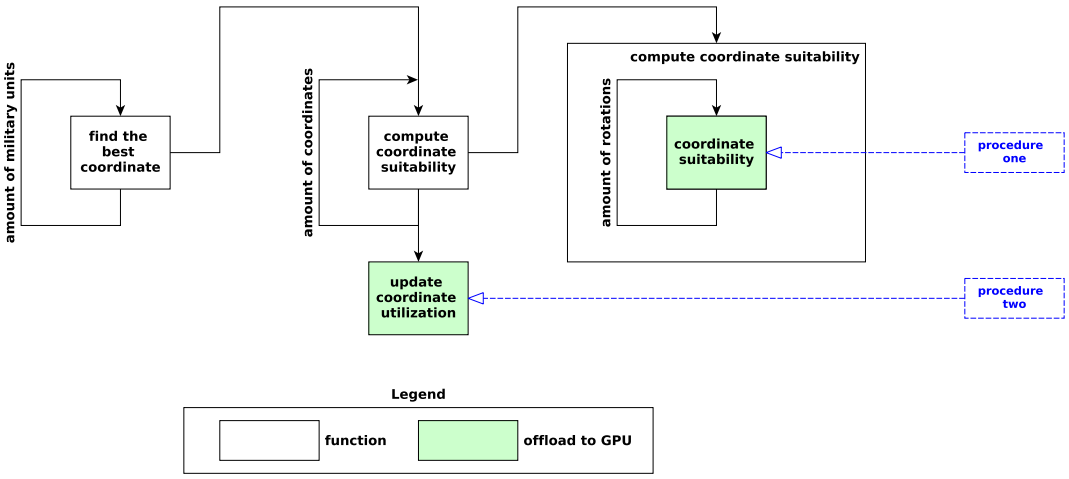


Figure 3.7 – MS Benchmark Stage `C`'s flowchart.

(e) We implement a binary tree parallel reduce algorithm for combining results from each thread block [NVI12]. The result of each thread block is the most suitable coordinate for the military unit assigned to the thread block.

(f) A single thread of the thread block performs atomic operations to update the utilization status of the coordinate chosen for the military unit assigned to the thread block. An alternative strategy to update the status of the coordinate is using a traditional algorithm reduction. However, such a strategy has large memory requirements. Since MS is a streaming application, the continuous data flow imposes that the GPU will process several drones simultaneously. Thus, a strategy with large memory requirements is unsuitable because it makes the GPU run out of memory.

(g) We apply loop collapsing and unrolling to lower the number of branches in the GPU kernel.

(h) We launch a GPU kernel for each type of military unit. Each type of military unit performs different routines for verifying the suitability of a coordinate. Suppose the GPU threads from the same thread block perform different routines. In that case, the instructions between the threads will diverge, imposing a GPU underutilization.

In short, the resulting GPU kernel performs a sequence of two simple loops. In the first loop, the GPU threads compute the coordinates' suitability and perform $O(CN/n)$ branches, where CN is the number of coordinates, and n is the thread block size. In the second loop, the thread block performs a reduction algorithm, and each thread performs $O(log\ n)$ branches, where n is the thread block size. After the thread block finishes the reduction algorithm, a single thread from the thread block is selected. Then, the thread performs $O(1)$ atomic operations to update the status of the best

coordinate found. The total amount of threads performed by this strategy is the number of military units multiplied by the size of the tread blocks.

4. **Stage D**. Stage `D` is a lightweight stage performed by the GPU. The main characteristic of this stage is the use of the parallel pattern `Reduce` with a single output variable. This stage's objective is to validate the results found in stage C.

   Figure 3.8 illustrates Stage `D`'s flowchart. Rectangles are functions, and rectangles with a green background are functions offloaded to the GPU. Dashed rectangles are routines. Stage `D` is divided into three procedures. The first procedure is a single loop that counts the number of military units classified into each coordinate. The second procedure is a single loop that validates each military unit and counts the number of ones that passed the test. The third procedure compares the checksum reference value to the results from the first and second procedures. The Big O notation [CLRS09] of the stage `D` is $O(nc + nm + 1)$, where nc is the number of coordinates, and nm is the number of military units.

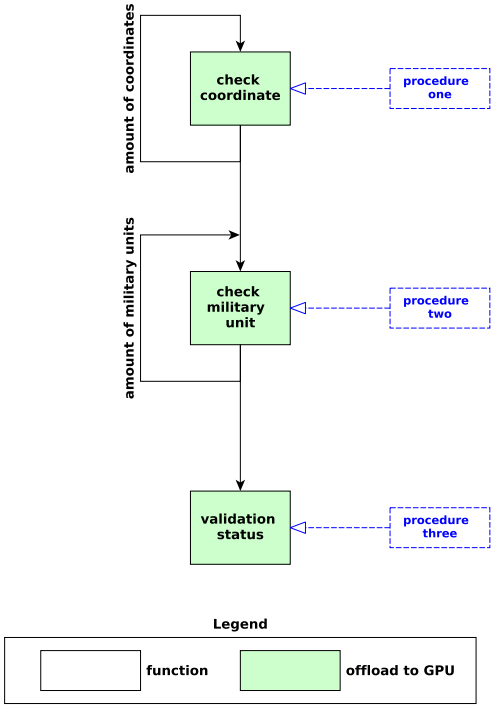


Figure 3.8 – MS Benchmark Stage `D`'s flowchart.

In the GPU implementation of the first procedure, we assign a GPU thread to each loop iteration. As this procedure is a parallel reduce pattern, we use atomic operations to combine the results from different thread blocks. However, only a single thread of each thread block performs the atomic operations (the thread updates the global reduce with the partial reduction of its thread block). A traditional parallel reduction implementation using buffers to store partial results is unsuitable for MS because parallelism increases memory utilization, resulting in application crashes. We applied the same parallel strat-

egy in the second procedure. The third procedure implements a checksum verification for validation status, and a single GPU thread executes it. If the drone passes the test, it assigns its status as `SUCCESSFUL`. Otherwise, it assigns its status as `UNSUCCESSFUL`. The MS benchmark execution is correct when all drones pass this test.

5. **Stage E**. Stage `E` is a lightweight stage performed by the CPU. The main characteristic of this stage is the CPU IO. The CPU simulates the output by writing the results of the stream element in the disk. The output is a configuration file containing a list of military units with their respective most suitable coordinates.

This section described the GPU implementation of the stream processing benchmark MS. In the next session, we report the limitations that we found in GSParLib and the optimizations that we provided to overcome each limitation.

3.4 Limitations and Improvements in GSParLib

This section discusses the limitations found in GSParLib's Driver and Pattern APIs. We also describe our strategies to overcome each one of them. We implemented the benchmarks presented in the above sections (CG, EP, FT, IS, MG, LD, MB, RT, and MS) to investigate GSParLib's performance and programmability.

We found five main limitations in the Driver API. The limitations can impact both performance and programmability depending on the application we want to apply the GPU parallelism. In the following items, we describe the limitations found and the optimizations we provided to overcome each one of them in the GSParLib's Driver API. The items are organized as follows:

1. **Different additional routines must be provided for CUDA and OpenCL**.

   - **Limitation**.

     When we create GPU kernels, we commonly need to define additional routines. The additional routines can be functions, variables, or constants used by a GPU kernel. In CUDA, we name them as device functions, device variables, and device constants [NVI20a].

     One of the main concepts of the GSParLib's Driver API idealized by Rockenbach [Roc20] is that the programmer must provide only the GPU kernel using CUDA or OpenCL syntax. However, GSParLib currently does not have any abstractions for the additional routines. The programmer must provide additional routines using CUDA and OpenCL syntax, which increases the programming effort and the knowledge required to exploit both programming models.

Moreover, writing an entire GPU kernel using CUDA or OpenCL syntax is already a drawback. A unified interface should not require any CUDA or OpenCL syntax from the programmer.

Code 3.1 shows an example of a GPU kernel that uses additional routines. The constant value named as `CONSTANT` has different type declarations for CUDA and OpenCL (lines 6 and 34), as well as the device function `routine` (lines 7 and 35). Considering the kernel source, several other differences are noted as the GPU kernel declaration (lines 10 and 38), global memory declaration (lines 11 and 39), shared memory declaration (lines 12 and 40), barrier for the thread block (lines 20 and 48), and information about the thread identifiers and the thread hierarchy (lines 14, 18, 22, 24, 26, 42, 46, 50, 52, and 54).

- **Improvement**. In order to overcome this limitation of programmability, we provided a set of abstractions for GSParLib. The abstractions implement a unified interface for CUDA and OpenCL.

  We created two mechanisms to provide a unified interface. First, we added a second typing in GSParLib to identify GPU kernels, device functions, and the memory hierarchy of the data. Second, we added functions that abstract CUDA and OpenCL routines, such as barriers or getting information about the thread hierarchy. Table 3.1 presents a summary of the abstractions added to GSParLib, which we can use with both Driver API and Pattern API. First column highlights if the abstraction is a keyword or a function. The second column describes the abstraction. The third column presents the signature or example of use for the abstraction.

  A GPU kernel inside GSParLib is a string, which allows extensive manipulation flexibility. In order to implement our abstractions, we created a preprocessing method for GPU kernels. GSParLib calls this method before compiling the string. To deal with the keywords, we created a routine that uses regular expressions to identify the GSParLib keywords in the source code and replace them with CUDA or OpenCL syntax. To deal with the abstractions that are functions, we implemented the low-level routines with CUDA and OpenCL. In the preprocessing method, GSParLib appends those routines at the beginning of the GPU kernel source. This way, the programmer can use the defined GSParLib functions to avoid a manual implementation of low-level routines with CUDA or OpenCL.

  Code 3.2 presents an example that uses our abstractions added to GSParLib that is equivalent to the previous code 3.1. Due to the abstractions, a single GPU kernel can be used to generate CUDA or OpenCL code; the single requirement is to import the C++ `headers` and `namespace` according to CUDA or OpenCL.

```
// CUDA version
#if defined(GSPARDRIVER_CUDA)
#include "GSPar_CUDA.hpp"
using namespace GSPar::Driver::CUDA;
std::string kernel_source = GSPAR_STRINGIZE_SOURCE(
const double CONSTANT = 56.64;
__device__ double routine(double a , double b){
  return a * b;
}
extern "C" __global__ void gpu_kernel(
  double* data){
    __shared__ double shared_data[1024];

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    double my_result = routine(data[index], CONSTANT);

    shared_data[threadIdx.x] = my_result;

    __syncthreads();

    if(threadIdx.x==0){
        for(int i=1; i<1024; i++){
            shared_data[threadIdx.x] += shared_data[i];
        }
        data[blockIdx.x] = shared_data[threadIdx.x];
    }
});
// OpenCL version
#else
#include "GSPar_OpenCL.hpp"
using namespace GSPar::Driver::OpenCL;
std::string kernel_source = GSPAR_STRINGIZE_SOURCE(
__constant double CONSTANT = 56.64;
double routine(double a , double b){
  return a * b;
}
__kernel void gpu_kernel(
  __global double* data){
    __local double shared_data[1024];

    int index = get_global_id(0);

    double my_result = routine(data[index], CONSTANT);

    shared_data[get_local_id(0)] = my_result;

    barrier(CLK_LOCAL_MEM_FENCE);

    if(get_local_id(0)==0){
        for(int i=1; i<1024; i++){
            shared_data[get_local_id(0)] += shared_data[i];
        }
        data[get_group_id(0)] = shared_data[get_local_id(0)];
    }
});
#endif
```

Code 3.1 – Example of different routines required when using GSParLib's Driver API.

Table 3.1 – Summary of the new abstractions for GSParLib.

Type	Description	Example of use / Signature
keyword	__gspar_device_function__ indicates that a function is a device function and will be called inside a GPU kernel.	__gspar_device_function__ additional_routine(double a, double b){return a*b;}
keyword	__gspar_device_kernel__ indicates that a function is a GPU kernel.	__gspar_device_kernel__ void gpu_kernel(__gspar_device_global_memory__int* array){...}
keyword	__gspar_device_global_memory__ indicates that the data is in the GPU global memory.	__gspar_device_global_memory__ int* array;
keyword	__gspar_device_shared_memory__ indicates that the data is in the GPU shared memory.	__gspar_device_shared_memory__ int shared_data[128];
keyword	__gspar_device_constant_memory__ indicates that the data is in the GPU constant memory.	__gspar_device_constant_memory__ double PI = 3.141592;
keyword	__gspar_macro_begin__ and __gspar_macro_end__ identify the beginning and the ending of a C/C++ macro declaration.	__gspar_macro_begin__ #define PI 3.141592 __gspar_macro_end__
function	__gspar_get_global_id__ returns the thread global id on the x, y, or z axis. The argument `axis` must be 0, 1, or 2.	__gspar_get_global_id__(int axis);
function	__gspar_get_thread_id__ returns the thread local id (position in the thread block) on the x, y, or z axis. The argument `axis` must be 0, 1, or 2.	__gspar_get_thread_id__(int axis);
function	__gspar_get_block_id__ returns the block id of the thread on the x, y, or z axis. The argument `axis` must be 0, 1, or 2.	__gspar_get_block_id__(int axis);
function	__gspar_get_block_size__ returns the block size of the thread on the x, y, or z axis. The argument `axis` must be 0, 1, or 2.	__gspar_get_block_size__(int axis);
function	__gspar_synchronize_local_threads__ performs a synchronization between the threads of the thread block.	__gspar_synchronize_local_threads__(void);
function	__gspar_atomic_add_int__ performs an atomic operation with integer numbers.	__gspar_atomic_add_int__(int* address, int value);
function	__gspar_atomic_add_double__ performs an atomic operation with float-pointing numbers of double precision.	__gspar_atomic_add_double__(double* address, double value);

2. **Macros are not recognized**.

   - **Limitation**. When programming with CUDA, the compiler automatically imports macros to GPU kernels. Among others, macros can include important parameters, define access patterns, and activate features of GPU APIs through pragma directives [NVI20a]. However, GSParLib cannot recognize macros, which impacts the programmability because it is impossible to activate some features, e.g., atomic operations from OpenCL. In our investigation, we found that the routine `GSPAR_STRINGIZE_SOURCE` from GSParLib is why the framework does not recognize macros.
   - **Improvement**. To solve this limitation, we provided two keywords for GSParLib (Table 3.1): 1) `__gspar_macro_begin__`, which identifies the beginning of a macro; 2) `__gspar_macro_end__`, which identifies the ending of a macro. The macro content uses the syntax of C/C++. The macros are manipulated in the preprocessing method before the GPU kernel compilation, as with the other abstractions.

3. **The compute capability of the GPU is not recognized**.

   - **Limitation**. The compute capability is a version number of GPUs from NVIDIA. It identifies the set of features supported by the GPU. When the GPU does not recognize the compute capability, it is impossible to use some CUDA features. For

```
// CUDA version
#if defined(GSPARDRIVER_CUDA)
#include "GSPar_CUDA.hpp"
using namespace GSPar::Driver::CUDA;
// OpenCL version
#else
#include "GSPar_OpenCL.hpp"
using namespace GSPar::Driver::OpenCL;
#endif
std::string kernel_source = GSPAR_STRINGIZE_SOURCE(
__gspar_device_constant__ double CONSTANT = 56.64;
__gspar_device_function__ double routine(double a, double b){
  return a * b;
}
__gspar_device_kernel__ void gpu_kernel(
  __gspar_device_global_memory__ double* data){
    __gspar_device_shared_memory__ double shared_data[1024];

    int index = gspar_get_global_id(0);

    double my_result = routine(data[index], CONSTANT);

    shared_data[gspar_get_thread_id(0)] = my_result;

    gspar_synchronize_local_threads();

    if(gspar_get_thread_id(0)==0){
        for(int i=1; i<1024; i++){
            shared_data[gspar_get_thread_id(0)] += shared_data[i];
        }
        data[gspar_get_block_id(0)] = shared_data[gspar_get_thread_id(0)];
    }
});
```

Code 3.2 – Example of the new abstractions for GSParLib.

example, atomic operations with double-precision floating-point numbers or co-operative groups (a feature that allows a global synchronization of GPU threads) require the compute capability information to work [NVI20a]. The more recent the compute capability, the larger the set of features available.

- **Improvement**. To overcome this limitation, we created a low-level routine that uses the NVIDIA CUDA Driver API [KH10, NVI20a] to collect the compute capability of the GPU. The compute capability is passed as an argument in the GPU kernel compilation, enabling all the GPU features.

4. **Limited use of atomic operations**.

   - **Limitation**. Atomic operations are important when programming GPUs because GPUs do not offer features for creating critical regions or semaphores for threads, except atomic operations. However, since GSParLib does not recognize the GPU compute capability, it is impossible to use the complete set of atomic operations from CUDA. In this case, when using the GSParLib for generating CUDA code, the programmer must manually implement atomic operations using the more basic features of CUDA. It increases the programming effort and the risk of introducing errors [NVI20a].

OpenCL does not offer a complete set of atomic operations such as CUDA. Then the programmers must always provide their implementation of atomic operations. Nonetheless, atomic operations do not work with OpenCL in GSParLib at all because the routines require the introduction of pragma directives that are not recognized by GSParLib. Moreover, implementing the atomic operations for CUDA and OpenCL are entirely different. Therefore, GSParLib should offer an abstraction for those operations.

- **Improvement**. We performed some modifications in GSParLib to fix these limitations.

  First, we created abstractions for atomic operations in CUDA and OpenCL, contemplating both integers (`gspar_atomic_add_int`) and floating-pointing numbers of double-precision (`gspar_atomic_add_double`), the signature of the functions can be checked in Table 3.1. Code 3.3 presents our low-level implementation of atomic operations for floating-pointing numbers of double precision. Lines 3 – 11 define the operation in CUDA, and lines 14 – 28 define the operation in OpenCL.

  Second, we enable the GPU resources. In CUDA, we identify and activate the compute capability of the GPU. Suppose the GPU compute capability is inferior to 6. In that case, we append the complete set of abstractions for atomic operations in the GPU kernel source because those routines are unavailable by default on those architectures [NVI20a]. Otherwise, the complete set of atomic operations from CUDA is available, and GSParLib just calls the available routines in CUDA. In OpenCL, we append our abstractions into the GPU kernel source and enable all OpenCL extensions in the GPU kernel compilation. In this case, we use the directive `#pragma OpenCL EXTENSION all: enable` for this purpose.

  In the NPB implementation, we used atomic operations in three benchmarks (EP, FT, and IS). As reported in our previous work [AGR+21], depending on the case, atomic operations can present better performance than algorithms of parallel reduction on modern GPU architectures.

5. **GPU thread hierarchy is not configurable**.

   - **Limitation**. It is not possible to specify the number of threads per block when programming a GPU kernel using the Driver API of GSParLib. However, this parameter can impact the GPU performance [AGR+21].

     Figure 3.9 illustrates an example of how changing the number of threads per block impacts GPU performance. We ran experiments with the function `q = A.p` from CG, varying the number of threads per block from the warp size (32) of the GPU up to the maximum threads per block supported by the GPU (1024). The x-axis presents the number of threads per block, and the y-axis presents the time in seconds. The blue curve presents the results of the benchmark CG running with the

```
#if defined(GSPARDRIVER_CUDA)
// implementation of atomic add with floating-pointing numbers of double precision in CUDA
__device__ double gspar_atomic_add_double(double* address, double val){
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val + __longlong_as_double(
    assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
#elif defined(GSPARDRIVER_OPENCL)
// implementation of atomic add with floating-pointing numbers of double precision in OpenCL
double gspar_atomic_add_double(__global double* valq, double delta){
    union {
        double f;
        unsigned long  i;
    } old;
    union {
        double f;
        unsigned long  i;
    } new1;
    do {
        old.f = *valq;
        new1.f = old.f + delta;
    } while (atom_cmpxchg((volatile __global unsigned long *)valq, old.i, new1.i) != old.i);
    return old.f;
}
#endif
```

Code 3.3 – GSParLib abstractions for atomic operations.

class B of NPB, and the green curve presents the results of class C. In this routine, the GPU performance decreases as the thread block size increases. When using 32 threads per block, the GPU kernel is 525% faster than 1024 threads per block. Tasks of different sizes are processed in the `q = A.p` GPU kernel. However, most tasks have a small size. In this case, the task size is the number of positions to be processed in an array. Thus, when there are more threads than positions to be processed, the threads stay inactive, imposing an overhead of creating idle threads and divergent instructions. The best number of threads per block to improve a GPU kernel's performance depends on the algorithm's characteristics and GPU hardware. Our previous work describes more details about how the aspects of an algorithm can impact the GPU performance and how they are related to the number of threads per block [AGR+21].

- **Improvement**. To overcome this limitation, we created attributes to store the number of threads per block in a GPU kernel (a GPU kernel is an object). The attributes are private and have public methods to allow the modification of their values. Table 3.2 presents the signature of the methods. The first column describes the abstraction. The second column presents the signature of the methods. The methods `setNumThreadsPerBlockForX`, `setNumThreadsPerBlockForY`, and `setNumThreadsPerBlockForZ`, allow configuring the number of threads per block on the axis x, y, and z.

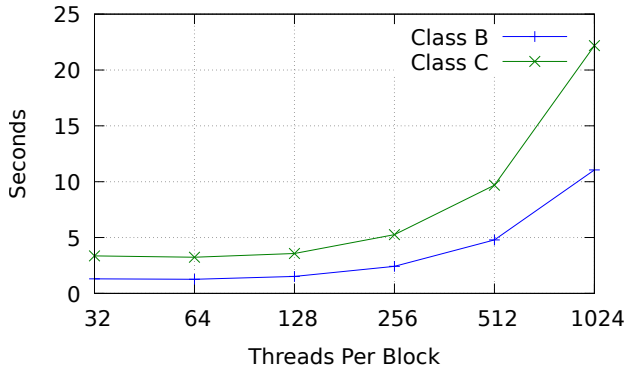


Figure 3.9 – Performance of the function `q = A.p` from CG when varying the number of threads per block (original source [AGR+21]).

Table 3.2 – Abstractions for configuring the number of threads per block in GSParLib.

Description	Example of use / Signature
setNumThreadsPerBlockForX sets the number of threads per block in the axis *x*.	void setNumThreadsPerBlockForX(unsigned long num);
setNumThreadsPerBlockForY sets the number of threads per block in the axis *y*.	void setNumThreadsPerBlockForY(unsigned long num);
setNumThreadsPerBlockForZ sets the number of threads per block in the axis *z*.	void setNumThreadsPerBlockForZ(unsigned long num);

If the programmer does not specify the number of threads per block, GSParLib uses the maximum number of threads per block supported by the GPU as configuration.

Now we discuss the limitations found in the Pattern API and describe our improvements to overcome each one. Pattern API inherits all Driver API issues and also inherits their improvements. Therefore, we describe the exclusively Pattern API limitations as follows:

1. **Memory transfers are always executed when a GPU kernel is executed**.

   - **Limitation**. Memory transfers between the CPU and the GPU are costly operations, and the good practice of GPU programming is avoiding them [NVI20a]. However, whenever the Pattern API executes a GPU kernel, GSParLib operates memory transfers. There is no option to keep the data on the GPU and reuse it.

     In robust computations like those present in NPB, where the application operates many GPU kernel launches, the excessive use of memory transfers becomes a critical problem for performance. When considering stream processing applications, memory transfers should be done only at the beginning of the first stage and at the end of the last stage.

   - **Improvement**. In the Pattern API, we manage the memory arguments of a GPU kernel through the method `setParameter`. GSParLib's original version only accepts host pointers as memory arguments. GSParLib creates an internal memory object for each host pointer before launching a GPU kernel and deletes them all when the GPU kernel finishes its execution.

To overcome this limitation, we overloaded the method `setParameter` to receive a GSParLib memory object. A few lines of code are enough to create a GSParLib memory object. Consequently, it does not negatively impact the programmability levels of GSParLib.

We also created another option to control the flow of memory transfers. The option is `GSPAR_PARAM_PRESENT`, which indicates that the data is already on the GPU and does not perform any memory transfers.

When using the method `setParameter` with a memory object and the option `GSPAR_PARAM_PRESENT`, the GPU kernel will use the data from the memory object instead of creating an internal memory object and performing memory transfers.

Creating a pointer or an object for the GPU memory is a common practice present in several GPU frameworks such as CUDA, OpenCL, and Thrust [KH10]. Providing arguments that indicate if the data is already present in the GPU is a practice adopted in high-level frameworks such as OpenACC [CJ17]. Memory objects from GSParLib can be used with any of the arguments to control the flow of memory transfers (`GSPAR_PARAM_IN`, `GSPAR_PARAM_OUT`, `GSPAR_PARAM_INOUT`, `GSPAR_PARAM_PRESENT`). Table 3.3 shows the signature of the overloaded method `setParamter`. The first column presents a description. The second column presents the signature.

Table 3.3 – Overloaded `setParameter` method to improve GSParLib memory transfers.

Description	Example of use / Signature
setParamter sets a memory object as a GPU kernel parameter. `T` is a generic data type. `name` is the name of the data inside the GPU kernel. `data_device` is the memory object that will be set as parameter in the GPU kernel. `direction` is the memory transfer direction for the memory object (it can be `GSPAR_PARAM_IN`, `GSPAR_PARAM_OUT`, `GSPAR_PARAM_INOUT`, or `GSPAR_PARAM_PRESENT`). `GSPAR_PARAM_PRESENT` is the option required to avoid memory transfers.	setParameter<T>(string name, MemoryObject device_data, enum direction);

Code 3.4 presents an example that uses the optimizations for memory transfers in the GSParLib. In the example are launched five GPU kernels that perform computations over the memory object `data_device`. Line 1 collects a reference from the GPU of ID `0`. Line 3 creates a memory object. Line 4 allocates GPU memory for the memory object and associates it to a host pointer. Lines 6 – 8 execute the first GPU kernel. The method `setParameter` in line 7 uses the argument `GSPAR_PARAM_IN` to copy the data to the GPU. Lines 10 – 20 launch the GPU kernels 2, 3, and 4. Those launches use `GSPAR_PARAM_PRESENT` as an argument, avoiding any memory transfers. Lines 22 – 24 launch the last GPU kernel, use `GSPAR_PARAM_OUT` as an argument, and transfer the data to the host after finishing the execution. Following this strategy, we only perform two memory transfers. In contrast, the GSParLib default version would require ten memory transfers.

```
1  auto gpu = driver->getGpu(0);
2
3  MemoryObject* data_device;
4  data_device = gpu->malloc(sizeof(int)*N, data_host);
5
6  Map* kernel_1 = new Map(kernel_source_1);
7  kernel_1->setParameter<int*>("data", data_device, GSPAR_PARAM_IN);
8  kernel_1->run<Instance>(dims);
9
10 Map* kernel_2 = new Map(kernel_source_2);
11 kernel_2->setParameter<int*>("data", data_device, GSPAR_PARAM_PRESENT);
12 kernel_2->run<Instance>(dims);
13
14 Map* kernel_3 = new Map(kernel_source_3);
15 kernel_3->setParameter<int*>("data", data_device, GSPAR_PARAM_PRESENT);
16 kernel_3->run<Instance>(dims);
17
18 Map* kernel_4 = new Map(kernel_source_4);
19 kernel_4->setParameter<int*>("data", data_device, GSPAR_PARAM_PRESENT);
20 kernel_4->run<Instance>(dims);
21
22 Map* kernel_5 = new Map(kernel_source_5);
23 kernel_5->setParameter<int*>("data", data_device, GSPAR_PARAM_OUT);
24 kernel_5->run<Instance>(dims);
```

Code 3.4 – Using memory transfer optimizations in GSParLib.

2. **The pattern `reduce` imposes a sequence of GPU kernel launches**.

   - **Limitation**. The current implementation of the pattern `reduce` available in the Pattern API launches a sequence of GPU kernels, where each GPU kernel computes a partial reduction. The larger the amount of data, the more GPU kernels are launched. However, launching several GPU kernels is expensive. Moreover, the current implementation of the pattern `reduce` requires synchronization between the CPU and the GPU before launching a new GPU kernel, which is also an expensive operation.
   - **Improvement**. To overcome this limitation, we provided an implementation of the pattern `reduce`, where a single GPU kernel executes the whole computation.

     Code 3.5 presents a CUDA implementation of the pattern `reduce` (a binary tree parallel reduce [KH10]) that is executed by a single GPU kernel. In this implementation, we assign a thread block to each subset of the data. In lines 3 – 9, we initialize the data. In line 8, we load the data on the GPU shared memory. In line 11, we synchronize the threads of the same block. In lines 13 – 18, we perform a partial reduction using the binary tree reduction algorithm and combine the results from the threads of the same block. In lines 20 – 22, we perform the global reduction by selecting a single thread of the thread block and writing the thread block's partial reduction in the global reduction using an atomic operation.

```
__global__ void single_gpu_kernel_reduce(int* data, int* result, int n){
    // initialization
    __shared__ int shared_data[THREADS_PER_BLOCK];
    int thread_global_id = blockIdx.x * blockDim.x + threadIdx.x;
    int thread_local_id = threadIdx.x;
    shared_data[thread_local_id] = 0;
    if(thread_global_id < n){
        shared_data[thread_local_id] = data[thread_global_id];
    }
    // thread block synchronization
    __syncthreads();
    // partial reduction
    for(int s = 1; s < THREADS_PER_BLOCK; s *= 2){
        if(thread_local_id % (2*s) == 0){
            shared_data[thread_local_id] += shared_data[thread_local_id + s];
        }
        __syncthreads();
    }
    // final reduction with an atomic operation
    if (thread_local_id == 0){
        atomicAdd(result, shared_data[0];
    }
}
```

Code 3.5 – Single kernel reduction with CUDA.

3. **Pattern Composition imposes a sequence of GPU kernel launches, synchronizations and memory transfers**.

    - **Limitation**. Conceptually idealized by Rockenbach [Roc20], the pattern composition from the Pattern API allows the composition of patterns. However, this feature currently executes a sequence of GPU kernels. It does not perform any transformations in the source code. Also, it performs synchronizations and memory transfers between each GPU kernel launch, which is an unnecessary overhead.
    - **Improvement**. As the use of the pattern composition is entirely optional, this is not a critical limitation. Additionally, the user can create a sequence of GPU kernels. This way, we will let the improvement of this feature as future work.

4. **No abstractions for using the GPU shared memory are available**.

    - **Limitation**. GPU shared memory is vital for cooperative work between the threads of a thread block or for reducing the memory latency of memory-bound computations. Currently, the Pattern API does not offer any abstractions for using the GPU shared memory, requiring low-level commands of CUDA or OpenCL.
    - **Improvement**. To overcome this limitation, we provided programming abstractions in GSParLib that allow allocating GPU data by specifying the memory hierarchy. In this case, we created the keyword `__gspar_device_shared_memory__` for allocating GPU shared memory data as presented in Table 3.1.

5. **Parameters of additional routines require OpenCL casts**.

   - **Limitation**. Suppose we pass arguments to additional routines of a GPU kernel and compile the GSParLib with OpenCL. If the parameter is data on the GPU global memory, GSParLib requires the use of casts with OpenCL syntax (`__global`). However, the programmer should not be responsible for providing such kinds of lower-level hacks using OpenCL syntax.
   - **Improvement**. Our new programming abstractions presented in Table 3.1 are enough to solve this limitation. Upon using the second typing for GSParLib, GSParLib generates OpenCL syntax before compiling the GPU kernel, eliminating the requirement of OpenCL details.

6. **GPU kernels are always compiled when the workload size is modified**.

   - **Limitation**. Another limitation of the Pattern API is that GSParLib always compiles a GPU kernel when the workload size is modified. In the NPB, it occurs in the benchmark MG. MG has irregular computations and launches each GPU kernel several times. However, it computes chunks of different sizes over the data. When GSParLib detects this behavior, it recompiles the GPU kernel, imposing a considerable slowdown in the GPU.
   - **Improvement**. In order to overcome this limitation, we modified the GSParLib to compile a GPU kernel a single time and allow its execution with different workload sizes. For this purpose, we use the workload size and the thread id to perform a verification. Suppose the thread's id is greater than the workload size; Then, this thread terminates its execution; Otherwise, the thread can perform the computations of the GPU kernel; This modification allows dynamically changing the workload size (without any recompilation). Compiling a GPU kernel once and reusing it with any workload size is a good programming practice allowed in CUDA and OpenCL [NVI20a, KH10].

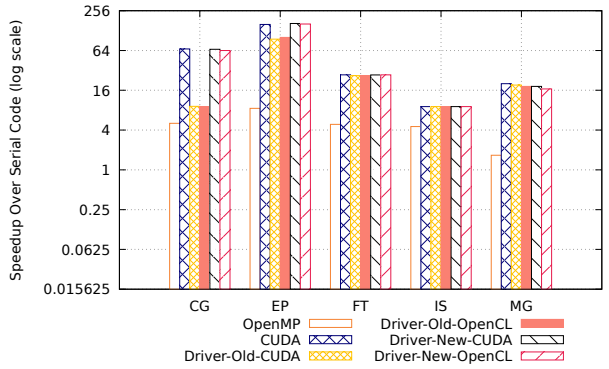
This section described the limitations that we found in GSParLib and the optimizations that we provided to overcome each limitation. In the next session, we present the performance evaluation of GSParLib on data parallelism benchmarks.

3.5 Impact on data parallelism with GSParLib

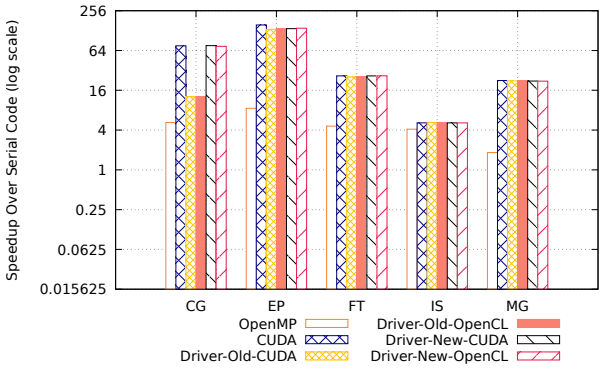
In this section, we evaluate the performance of the NPB programs using the original and improved versions of the GSParLib, CUDA, and OpenMP. We conducted the experiments in a computer equipped with Intel Xeon E5-2620 (6 cores and 12 threads) and

64 GB of RAM. The machine's GPU is a NVIDIA Titan X Pascal with 12 GB of VRAM and 3584 CUDA Cores. The operating system was Ubuntu 20.04 LTS. The software used was CUDA 11, GCC 9, OpenCL 3.0, and OpenMP 4.5. We used the compiler flag `-O3` to enable automatic compiler optimizations. We used classes B and C as workloads. NPB has other options for workloads. However, S, W, and A are too small, while D, E, and F are too large to fit the GPU memory. We executed each test ten times and collected the average time, and they presented a negligible standard deviation. We calculated the speedups concerning the execution time of the serial code on the CPU.

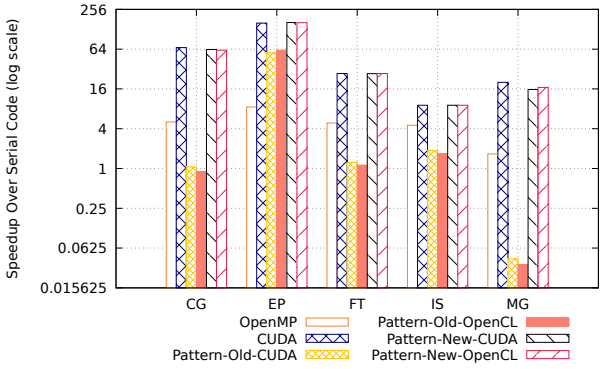
Figure 3.10 presents the speedup of the parallel implementations over the serial code on the CPU for the workloads B (Figures 3.10(a) and 3.10(c)) and C (Figure 3.10(b) and 3.10(d)). The x-axis lists the benchmarks evaluated. The y-axis presents the speedup over the serial code, using a logarithmic scale with base 2. Concerning the versions: 1) We tested an OpenMP version [LGM+21] (this version uses 12 threads, the number of logical cores of the processor) and a CUDA version [AGR+21]; 2) We tested all the combinations of the GSParLib Driver and Pattern APIs, with and without our improvements (we named these versions as `New` and `Old`, respectively), and with CUDA and OpenCL backends.



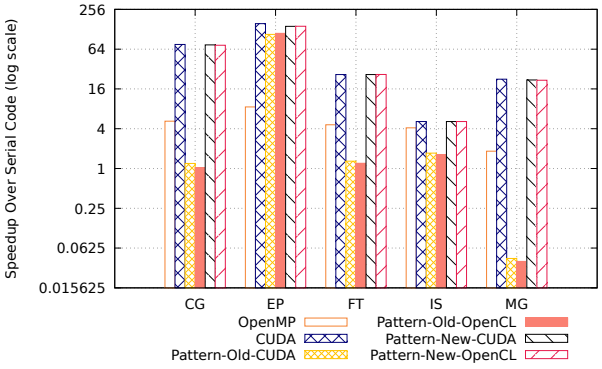
(a) Driver API comparison using class B.



(b) Driver API comparison using class C.



(c) Pattern API comparison using class B.



(d) Pattern API comparison using class C.

Figure 3.10 – Speedup of the NPB versions over the serial code on the CPU, using a logarithmic scale with base 2.

3.5.1 Impact on CG with GSParLib

In the CG benchmark, we can observe a performance limitation in `Driver-Old` due to the lack of allowing configuring the number of threads per block. CG performs better with small amounts of threads per block, like 32 or 64, while `Driver-Old` always uses 1024 threads per block. `CUDA` achieves up to 75 of speedup, while `Driver-Old` achieves up to 12 of speedup. However, `Driver-New` is able to achieve performance equivalent to `CUDA`. `Pattern-Old` presented a very low speedup, even slower than the serial code when compiled with OpenCL and running the class B. CG benchmark has iterative routines and calls several GPU kernels thousand times. The main overhead highlighted in this benchmark is the excessive amount of memory transfers performed in the `Pattern-Old` version. GSParLib's original version imposes memory transfers at each GPU kernel launch when using the Pattern API. That is the main reason for the performance degradation in CG. The lack of options to configure the number of threads per block also negatively impacts the performance. When we observe the `Pattern-New` version, the overheads are solved, and the performance is equivalent to `CUDA`.

3.5.2 Impact on EP with GSParLib

We also observed a performance degradation due to the number of threads per block in the EP benchmark. Similar to CG, EP performs better with a small number of threads per block, while GSParLib's original version uses the maximum threads per block supported by the GPU. `CUDA` achieved up to 158 of speedup, while `Driver-Old`'s best speedup was 135. `Driver-New` version was able to achieve up to 163 of speedup, which is even better than `CUDA`. The NPB CUDA version uses dynamic shared memory [AGR+21], while our GSParLib version of NPB uses static shared memory. For this reason, depending on the benchmark, a parallel version can achieve a better performance than others. In the `Pattern-Old` version of EP, we observed a moderate performance degradation compared to the CG version. EP launches a single GPU kernel, while CG performs thousands of GPU kernel launches (GSParLib performs memory transfer at each GPU kernel launch). However, the memory transfers required in EP were still noticeable in the overall performance. Similar to `Driver-New`, `Pattern-New` presented a performance even better than `CUDA`.

3.5.3 Impact on FT with GSParLib

Unlike CG and EP, the GPU thread hierarchy does not largely impact the benchmark FT. This way, `Driver-Old` and `Driver-New` presented a performance equivalent to `CUDA`. `Pattern-Old` presented a significant overhead due to the excessive amount of memory transfers, achieving up to 1.29 of speedup. In contrast, `CUDA` achieved up to 27 of speedup. `Pattern-New` presented a performance equivalent to `Driver-Old`, `Driver-New`, and `CUDA`.

3.5.4 Impact on IS with GSParLib

The same overhead of performance from FT was observed in the IS benchmark when running the `Pattern-Old` version. IS benchmark launches several GPU kernels, and GSParLib's original version of Pattern API operates lots of memory transfers. In contrast, `Pattern-New` presented again a performance equivalent to `Driver-Old`, `Driver-New`, and `CUDA`. The number of threads per block predominantly impacts IS. However, IS performs better with a large number of threads per block. Once `Driver-Old` always chooses the maximum number of threads per block supported by the GPU, this version does not suffer any overhead.

3.5.5 Impact on MG with GSParLib

In the MG benchmark, we observed another overhead in the `Pattern-Old` version. `Pattern-Old` achieved a performance even worse than CG when running the MG benchmark. A GPU kernel is launched several times in the MG benchmark. However, the GPU kernels use a different workload size at each launch. GSParLib's original version always recompiles a GPU kernel when the workload size is modified. That is why we observe a considerable overhead. GPU kernel recompilations and memory transfers lead the GPU to very poor speedups in the `Pattern-Old` version. `Driver-Old`, `Driver-New`, and `Pattern-New` achieved a performance equivalent to `CUDA`. We observed a lower performance with class B. It occurs due to the differences in the memory allocation of the GPU shared memory.

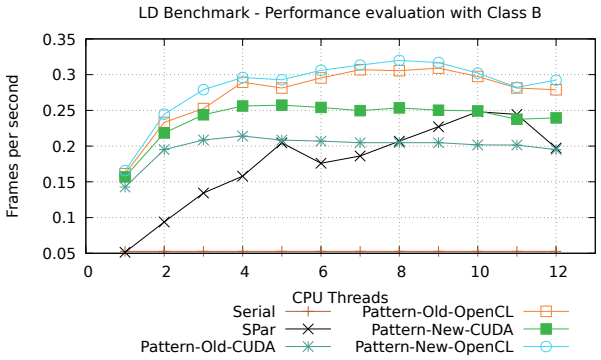
This section presented the performance evaluation of GSParLib on data parallelism benchmarks. In the next session, we present the performance evaluation of GSParLib on stream parallelism benchmarks.

3.6 Impact on stream processing with GSParLib

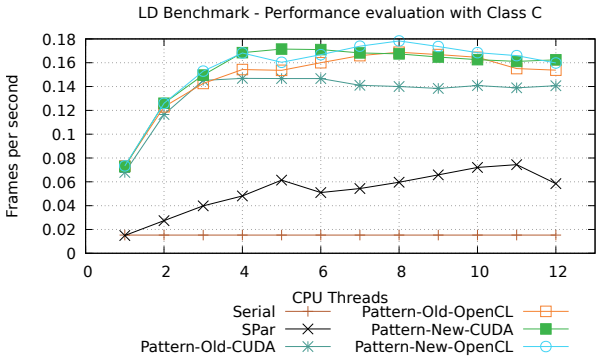
This section evaluates GSParLib's performance on stream processing benchmarks. We performed experiments using the same methodology and machine described in Section 3.5. As demonstrated in Section 3.5, GSParLib's Pattern API can achieve equivalent performance to GSParLib's Driver API. For this reason, we only tested the stream processing benchmarks with the Pattern API. We divided the workloads into two classes, B and C. The classes of each benchmark are composed of the following parameters:

1. **LD.B**. A 640*x*360 resolution video with 248 frames.
2. **LD.C**. A 1280*x*720 resolution video with 609 frames.
3. **MB.B**. A 1500*x*1500 matrix and 1500 iterations.
4. **MB.C**. A 2500*x*2500 matrix and 2500 iterations.
5. **RT.B**. 128 images of 640*x*360 resolution.
6. **RT.C**. 256 images of 1280*x*720 resolution.
7. **MS.B**. 2048 drones, 3072 coordinates per drone, 1024 military units per drone, a 2048*x*2048 land size.
8. **MS.C**. 2048 drones, 6144 coordinates per drone, 1536 military units per drone, a 2048*x*2048 land size.

Figures 3.11, 3.12, 3.13, and 3.14 present the amount of elements processed per second (throughput) in the benchmarks LD, MB, RT, and MS. The x-axis lists the number of CPU threads, where the number of CPU threads corresponds to the number of replicas of each stage (when applicable). For instance, we can replicate the stages B, C, and D in the MS benchmark 3.3. If we set the CPU threads as 2, the benchmark will create two replicas of the stages B, C, and D, totaling 6 CPU threads in those stages in addition to the thread in stage A and the one in stage E. The y-axis presents the throughput (elements per second). Concerning the versions: 1) We tested a serial version and a parallel multi-core version with SPar; 2) We tested all the combinations of the GSParLib Pattern API, with and without our improvements (we named these versions as `New` and `Old`, respectively), and with CUDA and OpenCL backends.

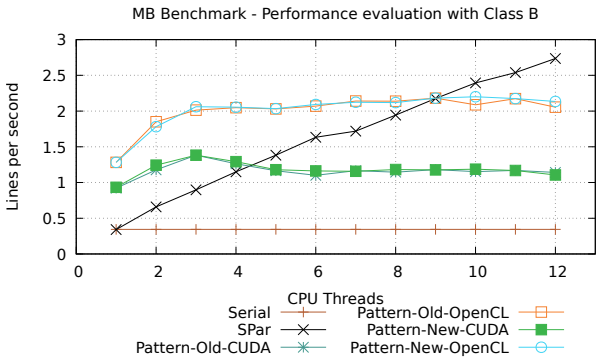


(a) Pattern comparison using the workload Class B.

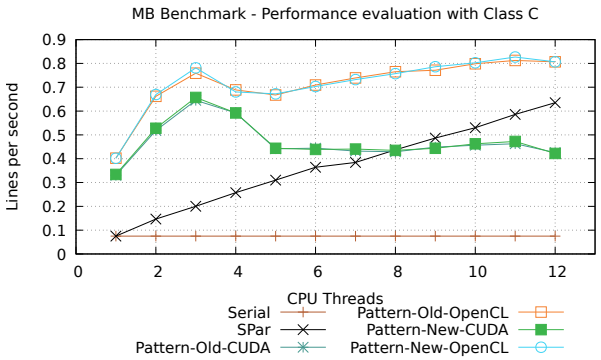


(b) Pattern comparison using the workload Class C.

Figure 3.11 – Throughput of LD benchmark versions.

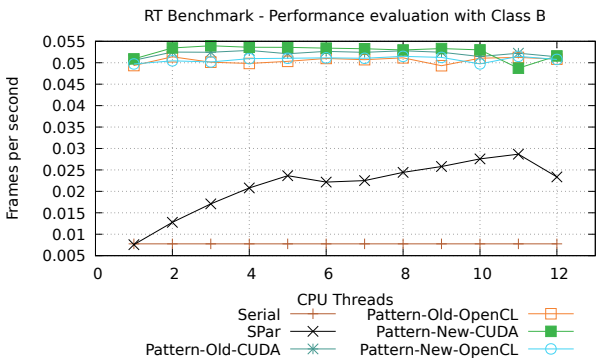


(a) Pattern comparison using the workload Class B.

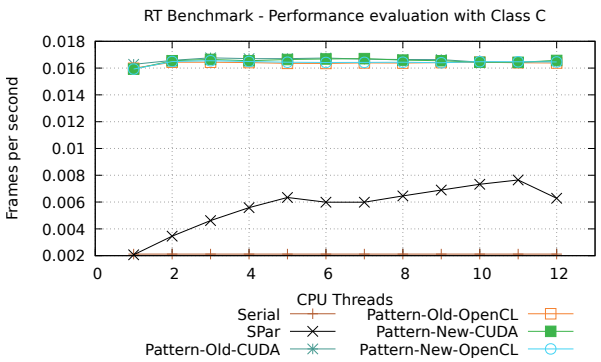


(b) Pattern comparison using the workload Class C.

Figure 3.12 – Throughput of MB benchmark versions.

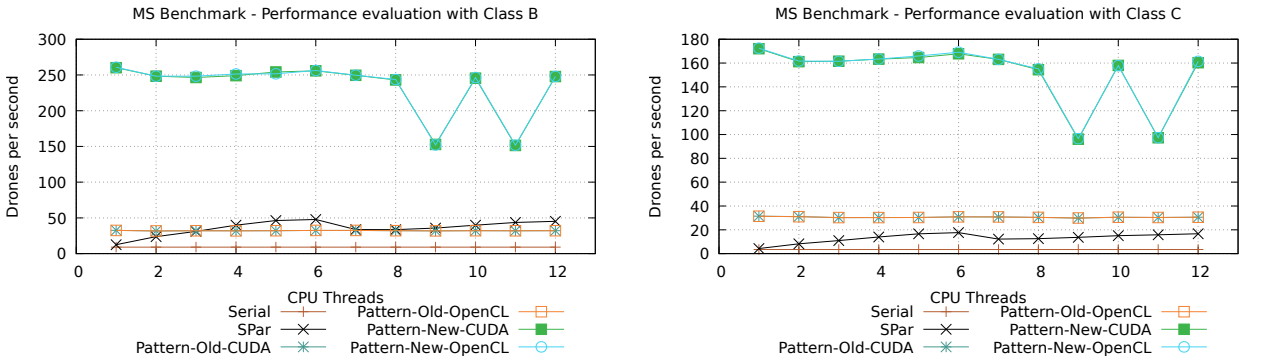


(a) Pattern comparison using the workload Class B.



(b) Pattern comparison using the workload Class C.

Figure 3.13 – Throughput of RT benchmark versions.



(a) GSParLib evaluation using the workload Class B. (b) GSParLib evaluation using the workload Class C.

Figure 3.14 – Throughput of MS Benchmark versions.

3.6.1 Impact on LD with GSParLib

In the LD benchmark (Figure 3.11), the GPU versions do not present a relevant improvement over the parallel multi-core version (`SPar`), except in the workload C. `SPar` version is up to 4.7 times faster than the serial version. In contrast, the GPU versions are up to 2.46 times better than `SPar` version. Varying the number of CPU threads in the multi-core version significantly impacts the throughput. In contrast, we observe a lower performance impact when varying the number of CPU threads in the GPU versions. Most computations are assigned to the GPU, imposing low CPU usage. When comparing the performance of `Pattern-Old-CUDA` and `Pattern-Old-OpenCL` to `Pattern-New-CUDA` and `Pattern-New-OpenCL`, poor performance improvements are observed. The reason is that this benchmark does not explore robust strategies or GPU resources. LD generates small loads for the GPU and imposes lots of communications between the CPU and the GPU. Those features are very different from the computations present in the NPB. NPB explores the GPU resources, imposes the use of robust strategies, generates large loads for the GPU, and avoids communication between the CPU and the GPU. In this case, the LD benchmark highlights that the new GSParLib's version does not present any performance advantages over GSParLib's original version when targeting simple applications. The same occurs in terms of programmability. The single GPU resource required in the LD benchmark is the global id of each thread. This resource is already available in the original version of the Pattern API. This way, this benchmark does not explore any of the resources that improved GSParLib's programmability.

3.6.2 Impact on MB with GSParLib

All LD limitations are present in the MB benchmark (Figure 3.12). However, MB generates even smaller loads for the GPU than to LD because computing a matrix row is less computing-intensive than processing a whole frame. Once it is not worth launching GPU kernels for non-compute-intensive routines, we do not observe a relevant performance improvement when using the GPU versions instead of the parallel multi-core version. Due to those MB features, we can not observe any performance and programmability improvements when comparing GSParLib's modified version to the original version.

3.6.3 Impact on RT with GSParLib

RT benchmark is similar to LD and MB (Figure 3.13). However, RT performs fewer communications between the CPU and the GPU than LD and has larger loads for the GPU than MB. Consequently, the GPU performance was better than the parallel multi-core version in classes B and C. However, RT is still a limited benchmark for evaluating GPU programming. As the GPU performs most computations, the number of CPU threads poorly impacted the throughput. As in LD and MB, the improvements are not noticeable when using GSParLib's new version over GSParLib's original version.

3.6.4 Impact on MS with GSParLib

In the MS benchmark, it is clear that the CPU `Serial` version (Figure 3.14) is very limited for computing the data collected from the drones. It can process only 9 drones per second with class B and 3 with class C. In contrast, `SPar` version that is a parallel multi-core implementation presents relevant results with up to 47 drones processed per second with class B and up to 17 drones processed per second with class C.

When considering GSParLib's original version (`Pattern-Old-CUDA` and `Pattern-Old-OpenCL`), the performance is very limited. The GPU is mostly similar or slower than `SPar` with class B and only slightly better than `SPar` with class C. While the MS Benchmark offers suitable routines for data parallelism. GSParLib's original version presents a limited performance because it requires memory transfers at each GPU kernel launch. In opposite, a suitable programming practice should perform memory transfers only in the stream's first and last stages, maintaining the data on the GPU. Another limitation is the number of threads per block that is not configurable and degrades the performance. MS GPU kernels consume lots of GPU resources. If we create large thread blocks, the

GPU's SMs will not have enough resources for launching several thread blocks in parallel, imposing a GPU under-utilization. Moreover, several GPU threads stay inactive in the irregular computations from stage `B` when the thread block is larger than the coordinate size. In short, GSParLib's old version is not able to offer relevant advantages over a parallel multi-core version on a robust stream application.

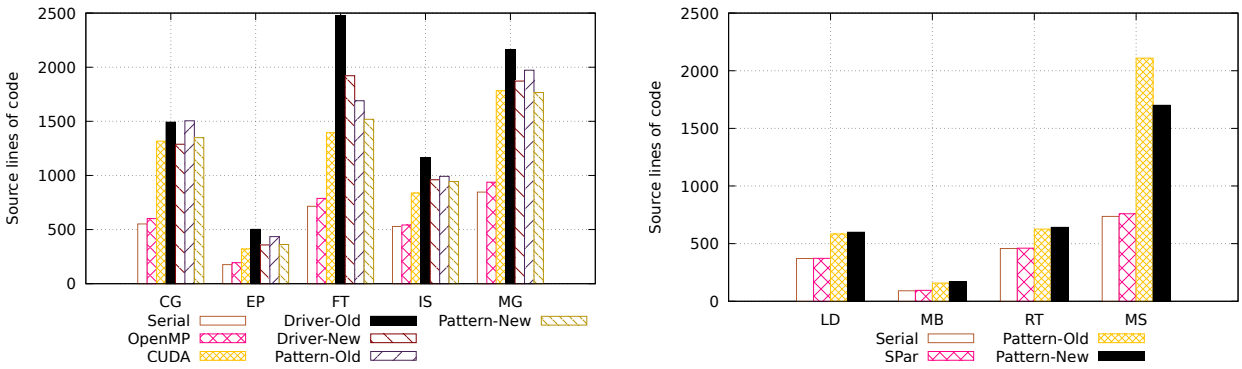
On the other hand, GSParLib's modified version (`Pattern-New-CUDA` and `Pattern-New-OpenCL`) was able to compute up to 260 drones per second with class B, and 172 drones per second with class C, which is a considerable improvement over the CPU versions (`Serial` and `SPar`). However, we must observe that the performance does not increase upon varying the number of CPU threads. The GPU performs most of the computations, and then the CPU always presents a low usage. A slowdown occurs when using 9 and 11 CPU threads due to the FastFlow runtime. FastFlow performs a thread pinning technique. It assigns a CPU thread to a physical core. This core always executes this thread, even if the core has to compute other tasks and other physical cores are idle. In this case, the thread pinning imposes a performance degradation. It occurs only in some cases because FastFlow statically distributes the CPU threads among the physical cores of the processor. In the MS benchmark, FastFlow's algorithm performs a bad choice only when using 9 and 11 CPU threads.

This section presented the performance evaluation of GSParLib on stream parallelism benchmarks. In the next session, we discuss programmability aspects of GSParLib.

3.7 Impact on programmability with GSParLib

This section discusses aspects of programmability from the benchmarks tested with GSParLib. Figure 3.15 presents the number of source lines of code of each benchmark implemented. Figure 3.15(a) presents the results for the benchmarks of data parallelism. Figure 3.15(b) presents the results for the benchmarks of stream parallelism. The x-axis lists the benchmarks, and the y-axis presents the number of source lines of code. We used the software `SLOC` [Whe] to collect this metric. Several techniques help to evaluate the programmability of an API. Collecting the source lines of code is a simple metric that helps give a general idea of the programming effort of an API through the necessary code intrusion [Roc20]. However, we highlight that it does not represent code productivity.

Concerning data parallelism with the NPB programs (Figure 3.15(a)), our first observation is that `Driver-Old` and `Pattern-Old` versions require more source lines of code than `CUDA`. It occurs because we must write every GPU kernel and additional routines using both CUDA and OpenCL syntax. Moreover, in GSParLib's original version, we must manually provide low-level routines such as atomic operations. Finally, GSParLib requires that we manually include each macro in the GPU kernels. It also increases the source lines of



(a) Data parallelism benchmarks.

(b) Stream parallelism benchmarks.

Figure 3.15 – Source Lines of Code of each Benchmark tested.

code. In opposite, CUDA automatically imports every macro from the C/C++ program and offers abstractions to features such as atomic operations. In future work, we could provide the automatic import of macros to GSParLib.

In the versions `Driver-New` and `Pattern-New`, there are fewer lines of source code compared to `Driver-Old` and `Pattern-Old`. This reduction is because the improvements done in the GSParLib allow writing a single GPU kernel or additional routine that can work with both CUDA and OpenCL. With these improvements, the amount of source lines of code is more similar to the `CUDA` versions.

Concerning the benchmarks of stream parallelism (Figure 3.15(b)), all the observations relative to the NPB programs are valid to the MS benchmark. `Pattern-New` requires fewer source lines of code than `Pattern-Old` because we can provide a single routine for both CUDA and OpenCL. LD, MB, and RT benchmarks are more straightforward than MS because they do not explore GPU resources or robust strategies. Thus, we do not observe a relevant difference in the source lines of code when using `Pattern-Old` or `Pattern-New`.

The main objective of the programmability improvements is not to require fewer source lines of code but to provide a unified interface for both CUDA and OpenCL. Providing a single source code that can work with CUDA and OpenCL and run on different vendors' GPUs is a considerable advantage.

A recurrent limitation present in frameworks that provide abstractions is performance degradation as a higher level is the abstraction. However, GSParLib is now presenting a performance very similar to CUDA in both Driver and Pattern APIs.

Another limitation exhibited by APIs that provide programming abstractions is that they are susceptible to a lack of expressiveness. For instance, OpenACC does not allow synchronizing the threads of a thread block. The lack of features can impose strategies that are not suitable for GPUs and result in performance degradation. In this case, GSParLib also provides a large set of features, making its APIs flexible and powerful as it can now recognize the GPU resources.

Some high-level APIs also introduce other concepts and other programming models, increasing the difficulty of applying parallelism in a given application. This kind of limitation occurs with frameworks such as Thrust from NVIDIA [DSU20]. In opposite, the GPU programming community well knows CUDA nomenclature and concepts. CUDA is the standard way to exploit parallelism on NVIDIA GPUs, and NVIDIA is the leader GPU manufacturer. For those reasons, in the GSParLib we keep the nomenclature and concepts similar to CUDA.

This section discussed programmability aspects of GSParLib. In the next session, we present our final remarks about GSParLib.

3.8 Final remarks about GSParLib

In this chapter, we presented a study for evaluating GSParLib's performance and programmability. Table 3.4 summarizes the results by presenting the performance improvement of the optimized version of the GSParLib over the original version and CUDA. The first column indicates the API used. The second column indicates the benchmark used. The third column indicates the parallelism exploited (data or stream parallelism). The fourth column indicates the metric measured (speedup or throughput). The fifth column indicates the percentage of performance improvement of the GSParLib's modified version over the GSParLib's original version. The sixth column indicates the percentage of performance improvement of the GSParLib's modified version over CUDA.

Table 3.4 – Performance improvement of the optimized version of GSParLib over the original version and CUDA.

API	Benchmark	Parallelism	Metric	Improve over original version	Improve over CUDA
Driver API	CG	Data	Speedup	**629.32%**	0.59%
Driver API	EP	Data	Speedup	72.61%	**3.92%**
Driver API	FT	Data	Speedup	4.02%	0.42%
Driver API	IS	Data	Speedup	0.00%	0.00%
Driver API	MG	Data	Speedup	0.00%	1.00%
Pattern API	CG	Data	Speedup	6,135.29%	-1.47%
Pattern API	EP	Data	Speedup	188.75%	**1.92%**
Pattern API	FT	Data	Speedup	2,095.97%	0.00%
Pattern API	IS	Data	Speedup	385.03%	0.00%
Pattern API	MG	Data	Speedup	**54,500.00%**	-3.00%
Pattern API	LD	Stream	Throughput	21.05%	-
Pattern API	MB	Stream	Throughput	0.00%	-
Pattern API	RT	Stream	Throughput	0.00%	-
Pattern API	MS	Stream	Throughput	**718.43%**	-

Our investigation highlighted several limitations in the original version of both Driver and Pattern APIs. GSParLib's original version was impacted by different performance degradation, turning the GPU even slower than the serial code in different cases. In contrast, the results demonstrated that our improved version of GSParLib can achieve a performance equivalent to CUDA in representative and robust applications such as those in

NPB [XTC+14, DKO+19, BBB+94, AGDF20, AGR+21] (Figures 3.10(a) and 3.10(c)) and C (Figure 3.10(b) and 3.10(d)). Additionally, as we can see in Table 3.4; When we use the optimized version of GSParLib on data parallelism, the optimized Driver API is to up 629.32% better than the original Driver API (CG case). At the same time, the optimized Pattern API is up to 54,500.00% better than the original Pattern API (MG case). On the other hand, when we use the optimized version of GSParLib on stream parallelism, the optimized Pattern API is up to 718.43% better than the original Pattern API (MS case). That performance difference between data and stream parallelism is because stream processing applications have inherent overheads such as allocating memory for each new stream element. Finally, we observe poor performance improvements over LD (21%), MB (0.00%), and RT (0.00%) cases because those benchmarks are not robust stream applications. When comparing GSParLib to CUDA, GSParLib presented up to 3.0% of performance degradation. It occurs because upon using GSParLib's abstractions, the GPU threads perform more instructions than using CUDA directly. In contrast, GSParLib was faster than CUDA in a few cases. In these cases, the differences occur due to the shared memory allocation in the GPU. In the GPU implementation of the NPB, we used static shared memory with GSParLib and dynamic shared memory with CUDA. This CUDA version uses dynamic shared memory because this implementation allows modifying the thread hierarchy during the execution time [AGR+21].

GSParLib's original version offers a unified interface for the CPU routines for manipulating GPU resources, such as allocating GPU memory or launching a GPU kernel. At the same time, the programmer must provide the GPU routines in CUDA and OpenCL syntax. In opposite, our improvements over GSParLib's programmability provided a unified interface for CUDA and OpenCL, eliminating the requirement of low-level routines and hacks of those frameworks. This feature facilitates programming the GPU for non-specialists in both CUDA and OpenCL. Also, it allows code portability between GPUs of different vendors.

Once we treated the most critical limitations of the GSParLib, future works can introduce new parallel patterns to GSParLib. Moreover, the unified interface will also facilitate adding new features for GPUs in the SPar. Generating code for a unified interface is more affordable than directly generating CUDA and OpenCL code.

In the next chapter, we present our study concerning SPar. We describe the main limitations that we found in SPar; the main improvements that we provided to overcome each limitation; and a performance evaluation contemplating the original and optimized version of SPar.

4. SPAR EVALUATION AND IMPROVEMENTS

In order to evaluate SPar's programmability and performance when targeting GPUs, we implemented the stream benchmarks previously presented in Section 3 (LD, MB, RT, and MS). Our implementation contemplates the GPU directive `Pure` presented in Section 2.5. Only the MS benchmark requires the use of a parallel reduce pattern. However, it requires a partial reduce while SPar's directive `Reduce` performs a global reduce. Thus, we do not consider SPar's directive `Reduce` in our strategy, as we explained in Section 3.3, it imposes an overhead in this benchmark.

While we addressed most of the GSParLib limitations (Section 3), in this section, we highlight SPar's GPU extension limitations and only describe possible strategies to overcome them. We generate parallel code using the SPar compiler and manually replace the GPU code with the one containing gsparlib's optimizations since it is not doable for the time we had in the Master Thesis. Our implementation is a semi-automatic parallel code generation approach that was enough to provide insights into the strategies created. For future work, we could simplify code generation for GPU in SPar's compiler since GSParLib is currently fully supporting a unified interface for CUDA and OpenCL. Generating code for a unified interface simplifies the compiler complexity of generating long code routines of pure CUDA and OpenCL code.

We organize the subsequent sections as follows: Section 4.1 highlights the main limitations found in the SPar's GPU extension and also describes possible improvements. Section 4.2 presents a performance evaluation. Section 4.3 verifies SPar's programmability impact. Section 4.4 describes the final remarks about SPar.

4.1 Limitations and Improvements in SPar

We discuss SPar's limitations in the following:

1. **No information about the CPU threads is stored**.

   - **Limitation**. CPU threads' information, such as id, is essential to allow different parallelism strategies or optimizations. For instance, when specific data is common to GPU kernels launched by a CPU thread. If we can access and store information about the CPU thread, we can access the thread id to store a Boolean type. We can use the Boolean type to indicate if it is the first stream item processed by the CPU thread. Then this CPU thread can transfer the data to the GPU only once in the first execution. Otherwise, the CPU will transfer data to

the GPU at every GPU kernel launch. In general, several optimizations require information about the CPU thread, such as id or partial results.

- **Improvement**. SPar generates FastFlow code for dealing with parallel multi-core code. In the FastFlow framework, we define the routine of a CPU thread as a C++ *class*. In this case, it is possible to add information and define methods that allow the programmer to access or modify the data.

2. **Excessive copies of GPU kernels**.

   - **Limitation**. GSParLib compiles a GPU kernel during the execution time. The optimized choice for running the same GPU kernel several times is reusing the same compiled GPU kernel. GSParLib's original version had a limitation of kernel recompiling that we solved, as we highlighted in Section 3.4. SPar's compiler generates a GPU code with a similar overhead. The generated code by SPar performs a complete copy of all GPU kernels of the application for each stream element. For instance, if the application receives ten million stream elements, SPar will perform ten million copies. SPar's original version defines the stream element as a C++ *class* that contains the GPU kernels from all stages of the stream pipeline. However, performing those copies for each stream element is an expensive operation.
   - **Improvement**. A strategy to overcome this overhead is creating a C++ *class* for each CPU thread. The CPU thread can store its ID, a Boolean type, and its list of GPU kernels, in this case, the GPU kernels that this CPU thread will execute. Then, the CPU thread can check if it is processing a stream element for the first time. If positive, the CPU thread performs a copy of the GPU kernels. If negative, the CPU thread does not perform a copy of the GPU kernels. When we use this optimization, the amount of copies is equivalent to the number of CPU threads. For example, suppose four CPU threads execute a stream application. In that case, SPar will perform only four copies of GPU kernels instead of a complete copy (a copy of every GPU kernel in the application) for each stream element.

3. **Excessive amount of memory transfers**.

   - **Limitation**. SPar always performs memory transfers when it launches a GPU kernel. We modified GSParLib to allow memory reuse (Section 3.4), but this feature is not supported in SPar.
   - **Improvement**. A solution for this limitation is providing an algorithm in the SPar compiler to analyze which data SPar can reuse. A common practice is to transfer data to GPU in the first stage of the stream and transfer data back to the CPU only in the last stage. If data is always read and never written, SPar can transfer it to GPU in the first stage and do not transfer it back to the CPU in the last

stage. Suppose a CPU thread uses shared data between all the stream elements processed. That means the CPU thread can only transfer the data to the GPU when processing its first stream element.

4. **Thread hierarchy is not configurable**.

   - **Limitation**. Although the number of threads per block can broadly impact the performance of a GPU kernel [AGR+21], SPar does not allow configuring the GPU thread hierarchy.
   - **Improvement**. SPar's attribute `Pure` indicates a GPU kernel. An attribute could be added to SPar to define the number of threads per block for each GPU kernel. For instance, we could combine the supposed SPar attribute for configuring the thread hierarchy with the attribute `Pure`. If we use the attribute `Pure` combined with the attribute `GPU_THREADS(32)`, SPar could create a GPU kernel with 32 threads per block. If we use the attribute `Pure` alone, SPar could create a GPU kernel with the default number of threads per block.

5. **Lack of atomic operations**.

   - **Limitation**. SPar currently does not support atomic operations, while atomic operations are essential to support different parallel strategies. A case where atomic operations are necessary is when we need cooperative work between threads of different thread blocks. However, there is not enough memory on the GPU. To allow cooperative work between threads of different thread blocks, we can create a buffer on the GPU global memory for each GPU thread or each thread block. As any thread of a GPU kernel can access a buffer on the GPU global memory, GPU threads can check or combine the results of other threads or thread blocks. Nonetheless, if there is insufficient memory on the GPU, a possible strategy to overcome this limitation is using atomic operations instead of creating buffers. For instance, assume that the GPU threads output an integer. Suppose we want to accumulate the results from all the threads of the GPU kernel. Instead of creating routines to manipulate buffers, we can use atomic operations for integers and manually combine partial results from the GPU threads. A similar case occurs in the MS Benchmarks, as discussed in Section 3.3.
   - **Improvement**. Considering we already added support to atomic operations in GSParLib, supporting atomic operations in SPar is possible by adding an attribute that generates a call to GSParLib's atomic functions.

6. **Lack of barriers**.

   - **Limitation**. CUDA and OpenCL allow the creation of barriers for thread blocks, and CUDA currently supports a barrier between the threads of a whole GPU kernel. SPar does not support any barriers, and it decreases its flexibility. To force

a synchronization of GPU threads in SPar, we must wait until the GPU kernel finishes its execution. Suppose an algorithm requires several points of synchronization between the threads of a thread block. In that case, we have to launch several GPU kernels, which will impose a significant performance degradation. Otherwise, if the API allows thread block synchronizing, a single GPU kernel launch is enough to complete the computations.

- **Improvement**. We could add two more attributes to the SPar language to overcome the lack of barriers. We could provide an attribute for a block synchronization (all threads from a thread block) and an attribute for a grid synchronization (all threads from a GPU kernel). Synchronization between the threads of a block is already available in GSParLib, and it works with CUDA and OpenCL. The synchronization between the threads of a GPU kernel is a feature only supported by CUDA. However, it is possible to provide a similar feature with OpenCL. In this case, we could launch a sequence of GPU kernels and apply a barrier between each GPU kernel launch. It is slower than the CUDA mechanism but permits a unified interface between CUDA and OpenCL.

4.2 Impact on stream processing with SPar

For evaluating SPar's performance on stream processing applications, we performed experiments using the same methodology and machine described in Section 3.5. We tested the legacy stream processing benchmarks (LD, MB, and RT) and the MS benchmark. Our goal is to verify how much our GPU optimizations presented in Sections 4.1 and 3.4 impact the performance of the SPar's extension for GPUs. Legacy stream processing benchmarks are typically not compute-bound and can not fully explore the available GPU resources. Therefore, they may present limitations for testing SPar and GSParLib optimizations. For this reason, we complemented our analysis with the MS benchmark. Additionally, to investigate different aspects of the SPar generated code, we provided three different scenarios for evaluating the MS Benchmark. We describe the MS scenarios as follows:

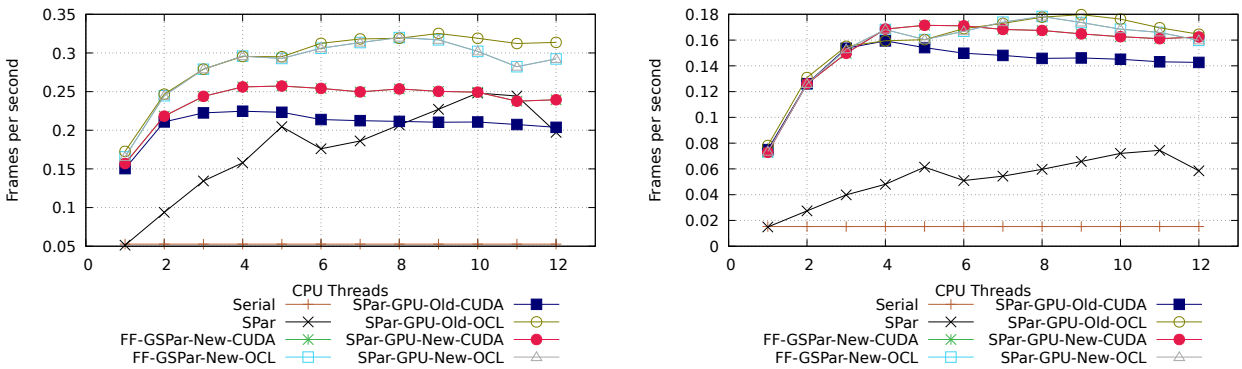
1. **MS Scenario 1**. We compare the performance of SPar's original code generated for GPU against a manual replacement of the GPU source code following the optimizations described in Section 4.1.

2. **MS Scenario 2**. We compare the performance of SPar's original code generated for multi-core against a manual implementation for multi-core using FastFlow and POSIX Threads (Pthread). As reported in Section 3.6.4, GSParLib presented a large performance overhead when combined with FastFlow. This scenario aims to verify how

much performance overhead is imposed by FastFlow against a low-level implementation using Pthread.

3. **MS Scenario 3**. This scenario aims to measure the GPU overhead when using different multi-core frameworks. Moreover, we verify if it is worth making SPar able to generate code for another multi-core framework when targeting GPUs.

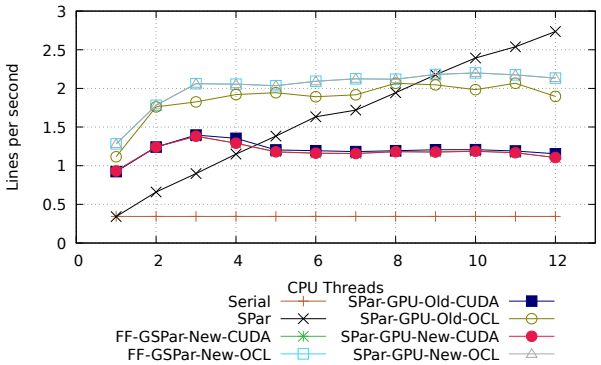
Figures 4.1, 4.2, and 4.3 present the performance results for the legacy benchmarks LD, MB, and RT. Figures 4.4, 4.5 and 4.6 present the performance results for the MS scenarios 1, 2 and 3. The x-axis lists the number of CPU threads, where the number of CPU threads corresponds to the number of replicas of each stage (when applicable). For instance, we can replicate the stages B, C, and D in the MS benchmark 3.3. If we set the CPU threads as 2, the benchmark will create two replicas of the stages B, C, and D, totaling 6 CPU threads in those stages in addition to the thread in stage A and the one in stage E. The y-axis presents the throughput (elements per second). Concerning the versions: 1) We tested serial and parallel multi-core versions with Pthread, FastFlow, and SPar; 2) We tested all the combinations of Pthread and FastFlow (we named these versions as `PT` and `FF`, respectively), using the GSParLib Pattern API (we named these versions as `GSPar`), with and without our improvements (we named these versions as `New` and `Old`, respectively), and with CUDA and OpenCL backends (we named these versions as `CUDA` and `OCL`); 3) We tested the SPar's extension for GPUs (we named these versions as `SPar-GPU`), with and without our improvements (we named these versions as `New` and `Old`, respectively), and with CUDA and OpenCL backends (we named these versions as `CUDA` and `OCL`).



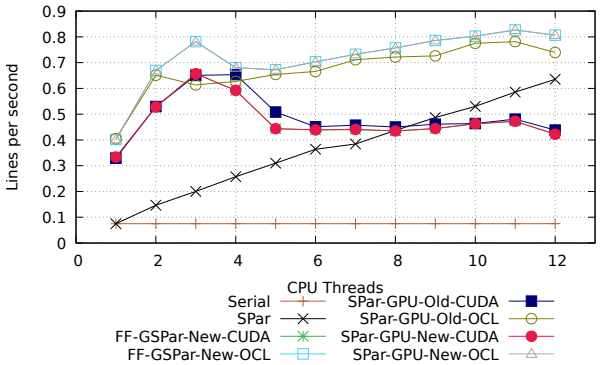
(a) SPar evaluation using the workload Class B.

(b) SPar evaluation using the workload Class C.

Figure 4.1 – SPar's GPU performance improvement on LD Benchmark.

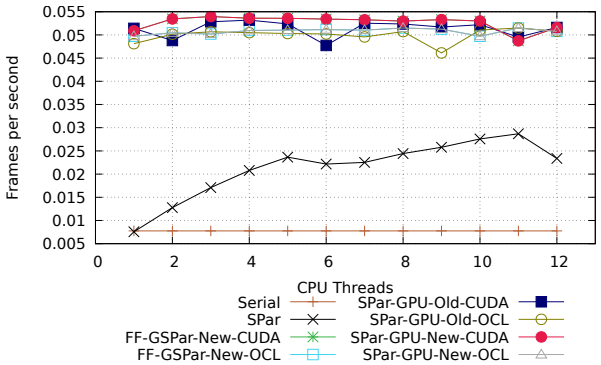


(a) SPar evaluation using the workload Class B.

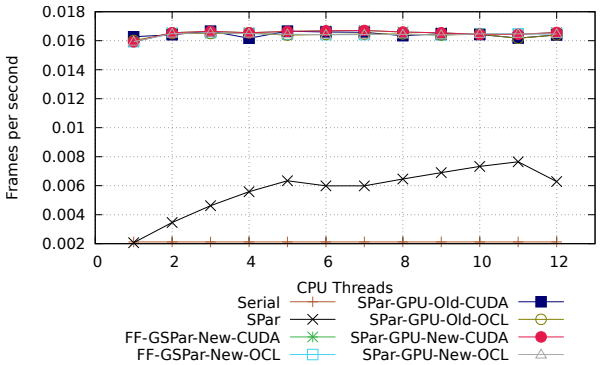


(b) SPar evaluation using the workload Class C.

Figure 4.2 – SPar's GPU performance improvement on MB Benchmark.

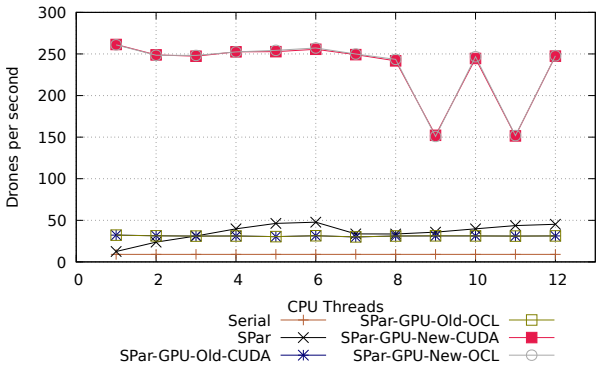


(a) SPar evaluation using the workload Class B.

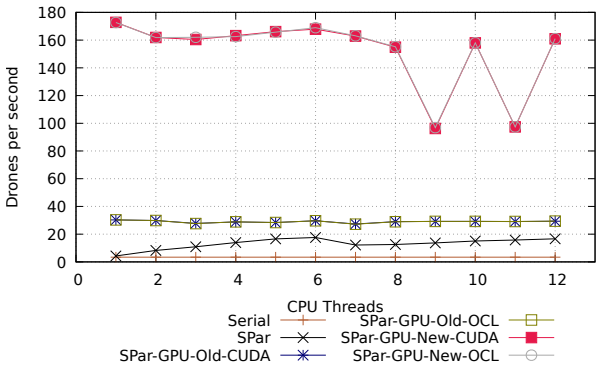


(b) SPar evaluation using the workload Class C.

Figure 4.3 – SPar's GPU performance improvement on RT Benchmark.

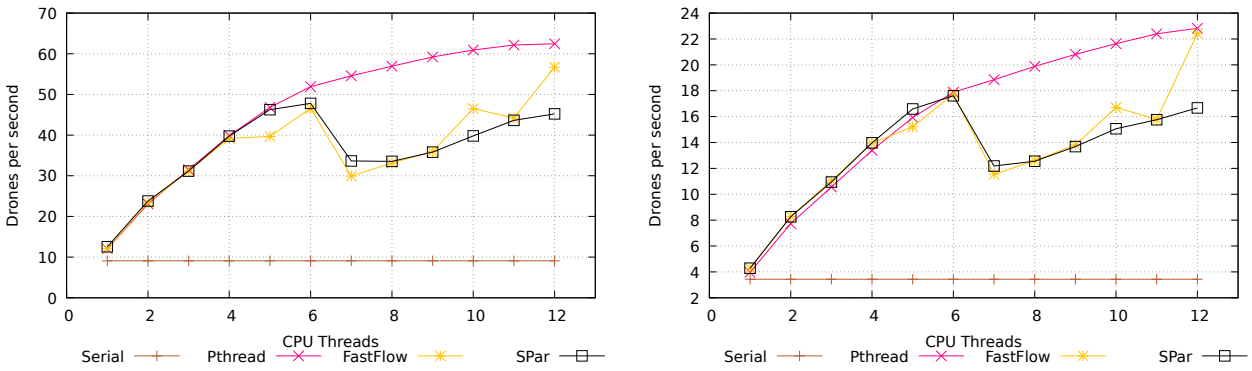


(a) SPar evaluation using the workload Class B.



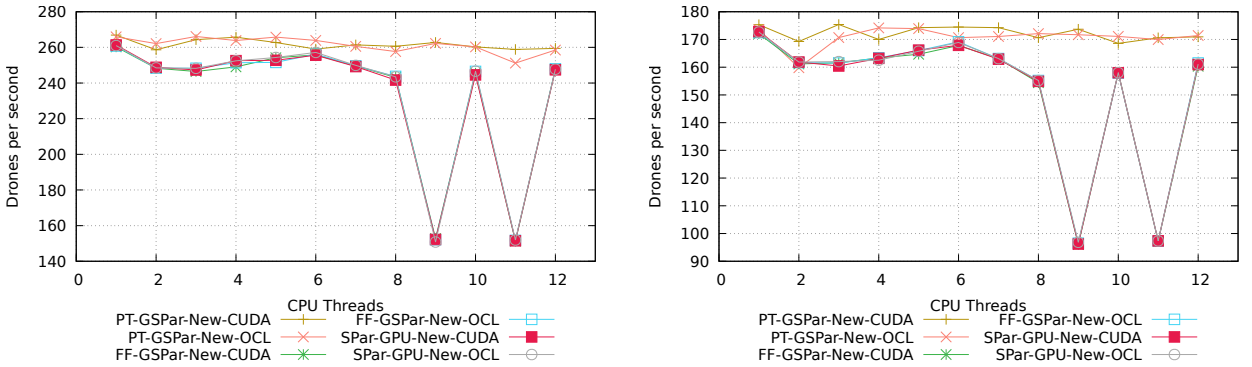
(b) SPar evaluation using the workload Class C.

Figure 4.4 – SPar's GPU performance improvement on MS Benchmark Scenario 1.



(a) SPar evaluation using the workload Class B. (b) SPar evaluation using the workload Class C.

Figure 4.5 – Comparison of multi-core versions on MS Benchmark Scenario 2.



(a) SPar evaluation using the workload Class B. (b) SPar evaluation using the workload Class C.

Figure 4.6 – Comparison of GPU versions on MS Benchmark Scenario 3.

4.2.1 Impact on LD with SPar

In the LD benchmark, `SPar-GPU-New-CUDA` and `SPar-GPU-New-OCL` were able to achieve a performance similar to the manual FastFlow implementation with the GSParLib modified version (`FF-Pattern-New-CUDA`, and `FF-Pattern-New-OCL`). On the other hand, `SPar-GPU-New-CUDA` presented only a small performance improvement over `SPar-GPU-Old-CUDA`, and `SPar-GPU-New-OCL` presented no performance improvement compared to `SPar-GPU-Old-OCL`. Although we provided substantial improvements to SPar, the LD benchmark is not computationally intensive. Thus, the application poorly exploits the GPU resources.

4.2.2 Impact on MB with SPar

The GPU implementations perform fewer GPU kernel copies in the MB benchmark than LD. However, the performance is still limited in all the GPU versions (Figure 4.2). `SPar-GPU-New-CUDA` and `SPar-GPU-New-OCL` achieved equivalent performance to FF-Pattern-New-CUDA and FF-Pattern-New-OCL. Nonetheless, they presented no performance improvement compared to `SPar-GPU-Old-CUDA` and `SPar-GPU-Old-OCL` as MB is also a non-computationally-intensive benchmark. OpenCL performed better than CUDA because GSParLib optimizes read-only data in OpenCL. In some cases, it can provide better performance than CUDA; that is the case of the MB benchmark. However, the multi-core parallel version outperforms the OpenCL versions when using more than 9 CPU threads in workload B. Additionally, the multi-core parallel version presents an almost linear speedup, which means that the performance could be potentially improved on a CPU with more cores. In the MS benchmark, we assign a GPU thread to each position of a matrix's line. As workload B generates small lines, several GPU cores stay idle, imposing a GPU underutilization. When using the workload C, the GPU versions `SPar-GPU-New-OCL` and `FF-Pattern-New-OCL` outperform the multi-core parallel version for each amount of CPU threads. In this case, as workload C generates larger loads for the GPU, the workload size improves the utilization of the GPU resources. For this reason, it is worth using the GPU in workload C and the CPU in workload B. GPUs require memory transfers and kernel launches for processing stream elements; They are costly operations that do not worth it when the workload is small, and the application is not computationally-intensive, as is the case of the MB benchmark.

4.2.3 Impact on RT with SPar

SPar's original and improved versions achieved similar results in the RT benchmark (Figure 4.3). RT is a simple benchmark. It only performs a single GPU kernel and does not fully exploit the GPU resources. Thus, this benchmark is not propitious to highlight the benefits of SPar's optimized code for GPUs.

4.2.4 Impact on MS Scenario 1 with SPar

We observe a considerable performance improvement when we compare SPar's optimized GPU code to the code generated by the original version of SPar (Figure 4.4). `SPar-GPU-New-CUDA` and `SPar-GPU-New-OCL` can compute up to 261 drones per second with workload B and 172 with class C. It is respectively 29 (Workload B) and 57 (Workload C)

times better than the serial version, which processes 9 (Workload B) and 3 (Workload C) drones per second. In contrast, `SPar-GPU-Old-CUDA` and `SPar-GPU-Old-OCL` compute up to 32 (Workload B) and 30 (Workload C) drones per second, which is mostly slower than the multi-core parallel version (`SPar`) with Workload B, and only ten times better than the serial version with Workload C. However, the optimized version of SPar's GPU code presents up to 44% of performance overhead when varying the number of CPU threads. The reason is that FastFlow's thread affinity mechanism imposes unbalanced computations for the CPU cores (we explained this FastFlow limitation in Section 3.6.4). In the `SPar-GPU-Old-CUDA` and `SPar-GPU-Old-OCL` versions, such overhead is not observable. Their bottleneck, caused by excessive memory transfers, is much more expensive and noticeable.

4.2.5 Impact on MS Scenario 2 with SPar

By isolating the multi-core parallel code in scenario 2 (Figure 4.5), we can better observe SPar's overhead when dealing with stream processing applications. The `Pthread` version performance improves as the amount of CPU threads increases. `Pthread` performance is up to 6 and 7 times better than the serial version. `FastFlow` has an unstable performance when the number of replicas is larger than the number of physical cores in the CPU. `SPar` multi-core generated code has a similar overhead. Performance overhead of `FastFlow` compared to `Pthread` when using hyper-threading is up to 53%, while SPar is up to 46%. However, after starting using hyper-threading, there is no case where `SPar`'s multi-core generated code can achieve a performance similar to `Pthread`, while `FastFlow` does.

4.2.6 Impact on MS Scenario 3 with SPar

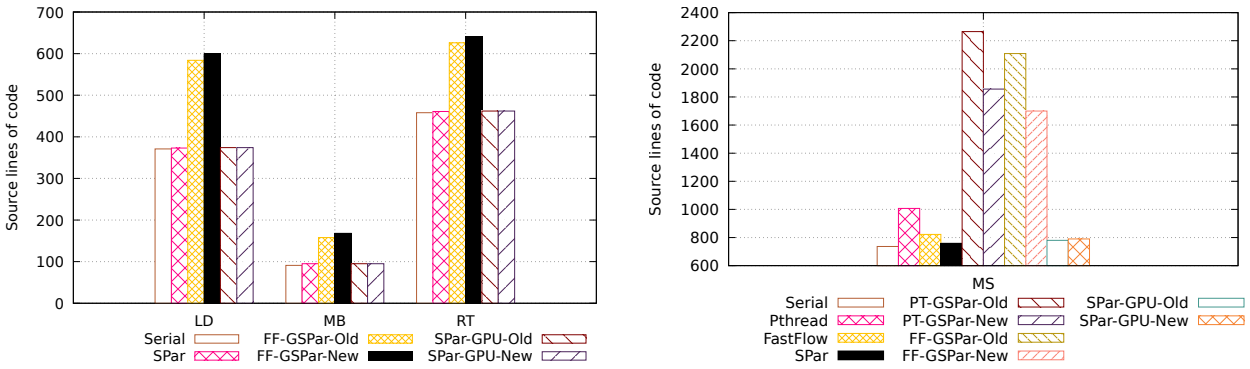
Scenario 3 presents the performance of SPar's optimized GPU generated code compared to a manual implementation using FastFlow combined with GSParLib, and Pthread combined with GSParLib(Figure 4.6). In scenario 2 (Section 4.2.5), a different performance occurs when using a manual implementation of FastFlow and the FastFlow code generated by the SPar compiler. Nonetheless, when targeting the combined parallelism from the CPU and GPU, no performance difference is observed between SPar's generated code compared to a manual implementation with FastFlow and GSParLib. The differences in performance between the FastFlow manual implementation and the FastFlow generated code are similar because the GPU performs most computations.

Regarding the performance differences between SPar's GPU generated code and Pthread combined with GSParLib, the best result of `Pthread-New-CUDA` and `Pthread-New-OCL` is only up to 3% better than the best result of `SPar-GPU-New-CUDA` and `SPar-GPU-New-OCL`.

However, the Pthread based versions did not present any overheads when dealing with stream elements' flow. In distinction, `SPar-GPU-New-CUDA` and `SPar-GPU-New-OCL` presented up to 45% of degraded performance when using hyper-threading. It occurs due to the FastFlow overhead caused by the thread affinity (we explained this FastFlow limitation in Section 3.6.4).

4.3 Impact on programmability with SPar

This section discusses aspects of SPar's programmability compared to the other frameworks tested. Figure 4.7 presents the results for each implemented benchmark. Figure 4.7(a) presents the results for the benchmarks LD, MB, and RT. Figure 4.7(b) presents the results for the MS benchmark's scenarios. The x-axis lists the benchmarks, and the y-axis presents the number of source lines of code. We used the software `SLOC` [Whe] to collect the source lines of code. This metric does not represent code productivity. However, it helps give a general idea about the effort of an API through the necessary code intrusion [Roc20].



(a) Legacy stream benchmarks.

(b) MS benchmark scenarios.

Figure 4.7 – Source Lines of Code of each Benchmark tested.

Concerning the multi-core versions, `Pthread` requires a manual implementation of each component necessary in a stream processing application, including the creation of queues for the communication between stages, functions for adding elements in the stages' queues, and techniques for dealing with the communications between different threads. `FastFlow` offers algorithmic skeletons for programming stream processing applications. This way, burdens such as creating queues for communications between stages or schedule mechanisms are unnecessary. However, the user still has to implement verbose routines, such as creating a C++ class for the computations of each stage. `SPar` largely improves the programmability over `Pthread` and `FastFlow` by automatically generating all routines necessary for a stream application. This way, `SPar` requires less programming effort than `FastFlow`, while `FastFlow` requires less effort than `Pthread`.

When dealing with GPUs, the programming effort is even more challenging. GSParLib facilitates GPU programming, but several mechanisms still must be provided. Additionally, exploring the parallelism of the CPU combined with the GPU is not an easy task. Due to these characteristics, the number of source lines of code of the GPU versions is up to 3 times larger than the serial version. The GPU versions with GSParLib require more significant programming effort than the `SPar` versions, even when programming the unified interface for CUDA and OpenCL of GSParLib's modified version. `SPar-GPU-Old` and `SPar-GPU-New` can successfully abstract the parallelism of both CPU and GPU. The amount of source lines of code of those two versions is slightly different because most GPU optimizations are internal to GSParLib.

4.4 Final remarks about SPar

This chapter presented a study for evaluating SPar's performance and programmability for both CPU and GPU. Table 4.1 summarizes the results by presenting the performance improvement of the optimized version of the SPar extension for GPUs over the original version. The first column indicates the API used. The second column indicates the benchmark used. The third column indicates the parallelism exploited (data or stream parallelism). The fourth column indicates the metric measured (speedup or throughput). The fifth column indicates the percentage of the performance improvement.

Table 4.1 – Performance improvement of the optimized version of the SPar extension for GPUs over the original version.

API	Benchmark	Parallelism	Metric	Improvement
SPar's extension for GPUs	LD	Stream	Throughput	14.52%
SPar's extension for GPUs	MB	Stream	Throughput	1.54%
SPar's extension for GPUs	RT	Stream	Throughput	0.00%
SPar's extension for GPUs	MS	Stream	Throughput	**771.68%**

Our investigation highlighted the main limitations of SPar's compiler's generated code. The results demonstrated that the generated code for the CPU has significant performance losses in controlling the flow of stream elements in a stream processing application. This limitation also interferes with the generated code for GPU. It leads to different overheads. The performance is up to 53% slower than using a lower-level API such as Pthread for providing the CPU routines.

Concerning the GPU, the original SPar's compiler generates GPU code with several performance costs, such as excessive amounts of memory transfers and copies of GPU kernels. Those overheads significantly decrease the GPU performance, presenting poor improvements over the serial or multi-core parallel implementations. When SPar utilizes the GSParLib's modified version, it obtains a considerable performance improvement on robust applications. As we can see in Table 4.1, the optimized version of the SPar's extension for

GPUs is up to 771,68% better than the original version in the MS benchmark. In contrast, we observe less significant improvements when testing the optimizations on non-robust applications, such as the benchmarks LD (14.52%), MB (1.54%), and RT (0.00%).

The main programmability limitation of SPar is the lack of functionalities such as allowing the programmer to configure the thread hierarchy, declare barriers or perform atomic operations. Other patterns should also be added to SPar to allow an efficient approach for different applications. Overcoming those limitations is crucial to improve SPar's flexibility to allow efficient parallelism on robust stream applications such as the MS benchmark. As SPar uses C++ attribute annotations, we can extend its functionalities without impacting the programming effort. Once our results demonstrated that Pthread achieved a more stable performance than FastFlow on stream processing applications. A future version of SPar's compiler could generate Pthread code for the CPU or provide means to solve the overheads present in the FastFlow.

The next chapter presents the related work concerning GSParLib and SPar. We briefly describe representative GPU frameworks based on structured parallel programming and code annotations. Additionally, we discuss works that also evaluated the NPB programs using GPUs.

5. RELATED WORK

The most popular framework for parallel programming in multi-core architectures is OpenMP [Kum02]. In contrast, the most popular frameworks for GPU accelerators are CUDA, OpenCL, and OpenACC [KH10]. OpenMP provides parallelism through high-level abstractions as directives. CUDA and OpenCL are low-level APIs that require a good knowledge about GPU programming from the programmers. OpenACC offers abstractions similar to OpenMP, but the programmer still needs to know details about the hardware and GPU programming techniques to achieve good performance [CJ17]. Although CUDA, OpenCL, and OpenACC are the most well-known frameworks for GPU programming, other GPU frameworks are available. As related work, we selected work directly related to SPar and divided them into two groups. The first group comprises GPU frameworks based on wrappers and structured parallel programming because SPar generates GPU code using GSParLib's interface. The second group comprises frameworks for heterogeneous platforms that can generate GPU code through annotations. We present representative work for each group in the following Sections 5.1 and 5.2. We also discuss the related work that provided the NPB with GPUs in Section 5.3.

5.1 Frameworks Based on Wrappers and Structured Parallel Programming

In this section, we briefly present GPU frameworks based on wrappers and parallel patterns that are related to GSParLib, which is the framework used by SPar to generate GPU code. SkelCL [SKG11] is a C++ framework that generates OpenCL code. It provides the patterns `map`, `reduce`, `zip`, `scan`, and `stencil`. To implement the parallel patterns, the programmer must create a class with the pattern desired, and use as parameter a string with the code that must run in parallel. The low level OpenCL code is then generated by SkelCL. The author published their latest paper about SkelCL in 2014. The SkelCL approach is very similar to GSParLib [Roc20].

Thrust [NVI] is a C++ template library integrated with CUDA and developed by NVIDIA. Thrust supports the parallel patterns `map` (named as transform), `reduce`, `stencil` and `scan`. Patterns are written as C++ STL functions and annotated with the keyword `__device__`, then each function is converted to a GPU kernel by the NVIDIA compiler. Although Thrust is a framework developed by NVIDIA, Thrust's programming model significantly diverges from CUDA, and its primary goal is to require minimal effort to write GPU programs. However, recent research about programmability highlighted that Thrust's interface and concepts increase the programming difficulties compared to CUDA [DSU20].

Like GSParLib, Boost.Compute [Lut] is a C++ framework that offers two APIs, a low-level API and a high-level API. The low-level API is an OpenCL wrapper, while the high-level API provides structured parallel programming. Boost.Compute support the patterns `map` (named as transform), `reduce`, `gather`, and `sort`. Like Thrust, to use each pattern from the high-level API, the programmer must implement an STL function.

FastFlow [APD+15] is a C++ framework based on parallel patterns that supports both stream and data parallelism. FastFlow offers several parallel patterns for CPUs such as `pipeline`, `farm`, `parallel for` (`map`), `parallel for reduce` (`map` and `reduce`), and `stencil`. For GPUs the single parallel pattern available is named as `loop-of-stencil-reduce`, which can be used to implement the patterns `map`, `reduce`, and `stencil`. Like Thrust and Boost.Compute, FastFlow also uses templates to provide the implementation of the parallel patterns. Similar to the concept of GSParLib, FastFlow can also generate code for CUDA and OpenCL. However, the programmer must provide the GPU kernel with pure CUDA and OpenCL syntax.

SkePU [ELK18] is another C++ library based on parallel patterns. It provides the parallel patterns `map`, `reduce`, `mapReduce`, `stencil`, `scan`, `mapPairs`, `mapPairsReduce`, and `mapOverlap`. The programmer must use C++ templates to apply parallelism using the parallel patterns available. SkePU can generate OpenMP code for CPUs or CUDA and OpenCL for GPUs. Unlike GSParLib, SkePU uses smart data containers and custom data structures resident on the host memory. When using accelerators like GPUs, the smart data containers automatically manage the memory device. They identify when to allocate memory or transfer data. The drawback of this approach is the need to provide a non-native C++ data structure. Another drawback is when the algorithm behind the smart data containers does not perform the best choice to optimize the memory transfers.

HIP is a C++ API that allows the programmer to use a single source code that can run on AMD and NVIDIA GPUs [AMD]. HIP generates Radeon Open Compute (ROCm) code for AMD GPUs and CUDA for NVIDIA GPUs. Analogous to CUDA, ROCm is a platform for AMD GPUs. HIP is similar to GSParLib's Driver API. It is a wrapper over GPU mechanisms, offers a unified interface for both GPU backends, and has no noticeable overhead compared to CUDA [KS19]. The syntax for writing a GPU kernel is equivalent to CUDA. In opposite to GSParLib's Pattern API, HIP does not provide any parallel patterns.

Kokkos is a C++ API that offers portability over different HPC platforms [TLGA+22]. Kokkos currently supports CUDA, HIP, SYCL, HPX, OpenMP, OpenMPTarget, and C++ threads as backends, and provides abstractions for parallel execution and data management. Thus, when using Kokkos, the user provides a single source that can run on different HPC platforms. However, the programmer must manually provide CUDA or HIP code when targeting a particular architecture such as a GPU. Kokkos provides three parallel patterns: `for`, `reduce`, and `scan`.

Table 5.1 presents general information about the related work compared to GSParLib. We sort the frameworks in alphabetical order, except the last one, GSParLib. Column `GPU Backend` indicates which GPU backend the framework supports. For example, Thrust only supports CUDA. Column `GPU Backend Abstraction` indicates if the framework requires low-level code from the backend. For instance, FastFlow requires GPU kernels using CUDA and OpenCL syntax. Column `Supported Patterns` lists all the GPU parallel patterns supported by the framework. Column `Stream Affordable` indicates if the GPU mechanisms required for programming stream applications are transparent to the user or facilitated through high-level abstractions. The main mechanisms are thread-safety for manipulating the GPU and asynchronous GPU kernel launches (commonly CUDA streams and OpenCL command queues). Others can be included, such as batch processing.

Table 5.1 – General Information about Frameworks based on Structured Parallel Programming.

Ref.	Name	GPU Backend	GPU Backend Abstraction	Supported Patterns	Stream Affordable
[Lut]	Boost.Compute	OpenCL	Yes	`Gather, Map, Reduce, Sort`	No
[APD+15]	Fast Flow	CUDA, OpenCL	No	`Map, Reduce, Stencil`	No
[AMD]	HIP	CUDA, ROCm	Yes	`Not available`	No
[TLGA+22]	Kokkos	CUDA, HIP, SYCL	No	`for, reduce, scan`	No
[SKG11]	SkelCL	OpenCL	Yes	`Gather, Map, Reduce, Scan, Stencil`	No
[ELK18]	SkePU	CUDA, OpenCL	Yes	`map, reduce, mapReduce, stencil, scan, mapPairs, mapPairsReduce, mapOverlap`	No
[NVI]	Thrust	CUDA	Yes	`Map, Reduce, Scan, Stencil`	No
[Roc20]	GSParLib	CUDA, OpenCL	Yes	`Map, Reduce`	Yes

Only GSParLib, FastFlow, and SkePU provide support to both CUDA and OpenCL (Table 5.1, column `GPU Backend`). However, the FastFlow developers discontinued the GPU support. CUDA is the standard form to exploit parallelism on NVIDIA GPUs. It provides many mechanisms for exploiting advanced GPU parallelism strategies, and NVIDIA is the main GPU manufacturer. However, CUDA only supports NVIDIA GPUs. On the other hand, although OpenCL offers fewer resources than CUDA, OpenCL allows exploiting the parallelism on GPUs of different vendors and even other accelerators. Thus, supporting CUDA and OpenCL is essential for the best performance and portability.

Most related work provide abstractions over CUDA and OpenCL low-level mechanisms, except FastFlow and Kokkos (Table 5.1, column `GPU Backend Abstraction`). When a GPU framework requires CUDA and OpenCL low-level routines, it increases the programming effort and the level of expertise necessary for programming the GPU.

GSParLib offers fewer parallel patterns than most related work because GSParLib is still an initial project (Table 5.1, column `Supported Patterns`). While the parallel patterns

`map` and `reduce` from GSParLib offer a considerable flexibility to approach different problems, a larger set of parallel patterns improves the programmability.

In contrast, the related work is not affordable for stream processing (Table 5.1, column `Stream Affordable`). 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 creates CUDA streams and OpenCL queues and manages them to allow asynchronous GPU kernels. These features are obligatory for approaching a stream processing application with GPUs. Additionally, GSParLib offers an abstraction for batch processing that is an optimization for stream processing on GPUs. 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.

5.2 Frameworks Based on Code annotations

This section presents frameworks based on code annotations that can generate code for heterogeneous architectures targeting GPUs.

OpenMP [Opeb] is a standard in the industry for multi-core CPUs, and it allows the application of parallelism in a source code through the use of directives. From version 4.0 onwards, OpenMP started supporting code generation for GPUs.

OpenACC [Opea] is a framework based on code annotations that generate GPU code. The basic directives available on OpenACC are similar to OpenMP. However, OpenACC has a more extensive set of directives and functionalities. It also exposes some low-level features of GPUs, such as thread hierarchy.

hiCUDA [HA11] is a framework developed by the academia that offers a high-level abstraction for CUDA using pragma directives. Each directive is attached to a specific CUDA functionality. Compared to OpenMP and OpenACC, hiCUDA exposes more GPU low-level mechanisms and increases the flexibility to approach different problems. On the other hand, hiCUDA requires knowledge about the concepts of CUDA programming, while OpenMP and OpenACC do not.

XscalableMP-ACC (XMP-ACC) [LTO+11] is an extension of XscalableMP (XMP) [Xca]. XMP is a framework that uses high-level abstractions similar to OpenMP but targets distributed parallelism instead. XMP-ACC was developed by the academia and added extensions to enable XMP to generate GPU code. Similar to hiCUDA, XMP-ACC only generates CUDA code, and the directives are similar to OpenMP and OpenACC.

XcalableACC (XACC) [NMS+14] is another extension of the XMP framework, but XACC targets the generation of GPU code with OpenACC instead of CUDA. Both exten-

sions are similar. They improve the programmability but still require GPU and distributed programming details, such as memory copies between CPU and GPU or between nodes.

Automatic Heterogeneous Pipelining (AHP) [PCR12] is aimed at pipeline applications, where the computations of a stage can be assigned to CPU cores or a GPU. When using AHP, the programmer must provide two parallel versions of the pipeline stages. One version must be optimized to run on CPU cores, and another must be optimized to run on the GPU. Once the programmer provides the parallel versions of the stages, the AHP automatically schedules the tasks between the CPU and the GPU. The literature names this programming model as hybrid programming [KH10, SE15, DR13].

Table 5.2 presents general information about the related work compared to SPar. We sort the frameworks in alphabetical order, except the last one, SPar. Column `GPU Backend` indicates which GPU backend the framework supports. Column `Annotation mechanism` informs the mechanism used by the framework to offer the functionality of annotations. Column `Simultaneous Parallelism` lists the capacity of simultaneous parallelism. For example, OpenMP supports running parallel code simultaneously on the CPU and GPU. Column `Architecture Abstraction` indicates if the framework abstracts details about the architecture targeted. For example, when using SPar, a programmer can offload code to a GPU without worrying about writing specific code to GPU. Column `Stream Parallelism Abstraction` indicates if the framework abstracts the stream parallelism.

Table 5.2 – General Information about Frameworks based on code annotations.

Ref.	Name	GPU backend	Annotation mechanism	Simultaneous Parallelism	Architecture abstraction	Stream Parallelism abstraction
[PCR12]	AHP	CUDA	Pragma	CPU, GPU	No	Yes
[HA11]	hiCUDA	CUDA	Pragma	GPU	No	No
[Opea]	OpenACC	Compiler dependent	Pragma	GPU	No	No
[Opeb]	OpenMP	Compiler dependent	Pragma	CPU, GPU	No	No
[NMS+14]	XACC	OpenACC	Pragma	GPU, Distributed	No	No
[LTO+11]	XMP-ACC	CUDA	Pragma	GPU, Distributed	No	No
[Roc20]	SPar	GSParLib	C++ attributes	CPU, GPU, Distributed	Yes	Yes

There are several advantages of SPar compared to the related work. As we can observe in Table 5.2 (column `GPU Backend Abstraction`), only SPar, OpenMP, and OpenACC offer portability between GPUs of different vendors by being able to generate CUDA and OpenCL code. The other frameworks only generate CUDA code. Concerning the mechanism of annotation, only SPar uses C++ attributes (Table 5.1, column `Annotation mechanism`). C++ attributes are part of the C++ language and inherit all its resources. Among the related work, SPar is also the single framework that offers the opportunity to explore three levels of parallelism, CPU, GPU, and distributed (Table 5.1, column `Simultaneous Parallelism`). Moreover, SPar is the single work that completely abstracts the architecture (Table 5.1, column `Architecture Abstraction`). The programmer 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. Finally, only AHP and SPar offer abstractions for exploiting stream parallelism (Table 5.1, column `Stream Parallelism Abstraction`). However, when using AHP, the programmer must manually provide the GPU kernels. In opposite, SPar automatically generates the GPU kernels through the C++ attributes.

Overall, SPar requires much less programming effort than the related work. SPar is unique at entirely abstracting the stream parallelism on the CPU combined with the data parallelism on the GPU. Moreover, SPar allows portability for GPUs of different vendors.

Nonetheless, SPar also presents limitations. OpenMP and OpenACC offer different directives for optimizing the serial code before applying the GPU parallelism. For instance, OpenACC offers directives like `loop collapse` that applies loop collapsing in the serial code and reduce the number of branch operations. Currently, SPar does not offer any attributes for GPU optimizations. Thus, the programmer must manually apply the transformations in the serial code before annotating the C++ attributes.

When we compare SPar to AHP, we can observe another limitation. AHP can assign a task to the CPU or the GPU through scheduling mechanisms. Depending on the application, this technique can provide substantial performance improvements. In contrast, SPar does not support hybrid parallelism yet.

5.3 NPB approaches with GPUs

The GPU research community has been widely using the NPB suite in recent years. Researchers use NPB for different purposes, such as testing a hardware optimization, a new compiler, or a new parallelism strategy. However, researchers commonly use an already parallelized version of the NPB to serve as a basis for continuing their research. Therefore, the main related work, the most utilized and cited ones, provided a parallel version of the NPB for GPUs using CUDA, OpenCL, or OpenACC.

In 2011, Seo et al. [SJL11] provided the first complete implementation of the NPB with OpenCL. In 2014, Xu et al. [XTC+14] provided a complete implementation of the NPB with OpenACC. Those two approaches had several limitations. For instance, the GPUs used for the implementation and experiments had a low memory capacity and a low number of cores, limiting the evaluation of performance and programmability. Concurrently, in 2020, Do et al. [DKO+19] provided optimizations for the implementation done by Seo et al. [SJL11], while we provided a CUDA version of the NPB and an analysis of the literature [AGDF20]. These two approaches are more recent and presented better results than previously due to optimized parallelism strategies and more powerful GPUs.

Porting the NPB for GPUs is challenging because it requires extensive refactoring and robust parallelism strategies. Thus, most of the literature uses those implementations mentioned above to conduct their research ([SJL11, XTC+14, DKO+19, AGDF20, AGR+21]). For instance, Kang et al. [KL20] evaluated our NPB CUDA version [AGDF20] in the context of edge device computing.

Nonetheless, none of the frameworks discussed in Section 5.1 (Boost.Compute, FastFlow, HIP, Kokkos, SkelCL, SkePU, and Thrust) were explored using the NPB. Given this context, our investigations with GSParLib contribute to evaluating the performance and programmability of structured parallel programming for GPUs when approaching robust computations such as those present in NPB. Concerning the frameworks based on code annotations, only OpenMP [STKC17] and OpenACC [XTC+14] were explored with the NPB suite. However, the performance evaluation is limited in the work presented by Shen et al. [STKC17]. The authors implemented the OpenMP 4.5 specification using the open-source OpenUH compiler. Although the NPB was able to be compiled with the authors' implementation, the performance was far worse than a manual implementation of NPB with CUDA and OpenCL.

6. CONCLUSION

In this work, we conducted a study to assess the performance and programmability aspects of GSParLib and SPar on representative applications. The GSParLib and SPar programming abstractions to support GPUs were in their early stages. They provided only basic features to exploit GPU resources and were previously evaluated only on smaller scenarios. Consequently, it was unknown whether GSParLib and SPar would be able to address robust computations and provide a competitive performance outcome.

Our investigations revealed that GSParLib and SPar have considerable limitations regarding performance and programmability. GSParLib was not able to recognize or enable GPU resources and required manual insertion of routines and hacks using CUDA and OpenCL programming models. In contrast, conceptually, GSParLib should offer a unified interface for both CUDA and OpenCL GPU backends. When we investigated the performance, we observed different overheads in the internal mechanisms of GSParLib. The main issues were excessive memory transfers, recompilation of GPU kernels, inefficient thread hierarchy, expendable GPU kernel launches, barriers, and branch operations.

Once SPar generates GSParLib code for GPUs, it inherits all the limitations present in GSParLib. Regarding SPar itself, its main limitations were the lack of functionalities to express parallelism. For instance, SPar does not support synchronizing GPU threads or performing atomic operations. This limited flexibility turns some parallelism strategies unfeasible to be implemented. Moreover, the CPU code generated by the SPar compiler for controlling the stream elements' flow also presented other limitations. The CPU's unstable performance highly impacts the GPU; in our experiments, it degraded the throughput performance by up to 50%. Another limitation was that the CPU performs excessive amounts of memory transfers and data copies.

Given these limitations, when approaching robust computations such as the ones present in NPB or the MS benchmark, GSParLib and SPar display huge performance losses. In some cases, the GPU performance was worse than the CPU's serial version.

In the second step of the study, we investigated methods and techniques to solve performance and programmability limitations on GSParLib and SPar. Our results culminated in a set of optimizations and improvements to GSParLib and a discussion about possible improvements to SPar. We summarize the main insights and achievements of this investigation as follows: 1) We presented a methodology to develop more agnostic frameworks independent of low-level programming interfaces; 2) GSParLib is now supporting a unified interface for CUDA and OpenCL; 3) GSParLib is now recognizing and providing abstractions for GPU resources such as shared memory and atomic operations; 4) GSParLib is now able to achieve equivalent performance to CUDA on robust applications due to new and optimized internal mechanisms. Therefore, GSParLib's updated version provided by this

work can approach more robust applications while delivering higher performance. Although we also provided optimizations to SPar, we did not modify SPar's compiler to generate the optimized GSParLib code automatically.

GSParLib and SPar have unique features among the available frameworks of the industry and academia. GSParLib is now fully supporting a unified interface for the standard APIs for GPU programming while presenting a performance equivalent to CUDA. The necessary mechanisms to exploit stream parallelism are transparent to the user. GSParLib automatically provides mutual exclusion sections when allocating or deleting GPU memory. GSParLib also automatically creates and manages CUDA streams and OpenCL command queues to allow asynchronous GPU kernels. GSParLib uses native C/C++ containers instead of requiring a new type of data and works with standard C/C++ compilers such as `g++` instead of requiring a separated compiler. On the other hand, SPar is a unique framework based on code annotations that generate code for stream processing targeting CPU and GPU, combining stream and data parallelism.

Once GSParLib's most critical limitations were treated by this work, other research opportunities remain open:

- **Provide new mechanisms to GSParLib.** Our optimized version of GSParLib can recognize and enable GPU resources of the underlying hardware. Also, we provided abstractions for functionalities such as atomic operations for integer and floating-point numbers. However, future works can add more abstractions to allow other GPU resources or mechanisms.

  For instance, NVIDIA recently introduced the *cooperative groups* feature. This mechanism allows the synchronization of GPU threads from different thread blocks. A traditional form to synchronize threads from different thread blocks in CUDA is launching a GPU kernel and wait it finishes its execution. However, GPU kernel launches and this kind of barrier are costly operations.

  Another recent feature from NVIDIA is CUDA Graph. CUDA Graph is similar to CUDA streams in functionality. It allows the programmer to launch asynchronous GPU kernels, which is required by stream processing applications. However, CUDA Graph considerably reduces the communication between the CPU and the GPU to manage the asynchronous GPU kernels. Consequently, CUDA Graph presents a substantial performance improvement over CUDA streams [HLLL22].

- **Provide new parallel patterns to GSParLib.** The results from this work demonstrated that the `map` and `reduce` patterns from GSParLib are flexible enough to approach robust computations such as those present in NPB. However, future work can add new parallel patterns to improve GSParLib's programmability. For instance, if the patterns `stencil`, `scan`, `gather`, and `scatter` are available in GSParLib's Pattern API, then the

programmer does not need to implement a variation of the `map` parallel pattern to use those functionalities, requiring less programming effort.

- **Provide support to other accelerators.** Although GSParLib supports only GPUs, the framework could support other many-core accelerators such as Field Programmable Gate Arrays (FPGA) and Intel Many Integrated Core (MIC). Like GPUs from AMD manufacture, the OpenCL can be used to exploit parallelism on other many-core accelerators.

- **Evaluate and improve support for multiple GPUs on GSParLib.** Currently, GSParLib's Driver API supports the use of multiple GPUs. The user can allocate memory or launch a GPU kernel by specifying the GPU id. However, future work can investigate if there is no overhead in the internal mechanisms of GSParLib when using multiple GPUs. Concerning GSParLib's Pattern API, the parallel patterns currently do not support multiple GPUs. GSParLib should be able to distribute the computations of a parallel pattern among several GPUs. Nonetheless, the current GSParLib version can only assign the computations to a single GPU.

- **Provide GPU runtime optimizations to SPar.** Future work can provide GPU optimizations to SPar that work during the execution time. For instance, SPar could automatically implement methods to vary the number of threads per block to process a GPU kernel and choose the best configurations as SPar processes new stream elements. Other important optimizations can be considered. Currently, SPar cannot process very large amounts of data such as petabytes. In this case, the GPU should not accept a new stream element except when there is enough device memory. Additionally, CPU threads keep copies of GPU kernels until the end of the application; if SPar no longer uses specific GPU kernels, they could be deleted by SPar to free memory. Those optimizations are pertinent for improving SPar's performance and flexibility to approach different stream applications that consume large volumes of data.

- **Provide a study of programmability with GSParLib and SPar.** In this work, we used the number of source lines of code to have a general idea of how GSParLib and SPar can impact the GPU programmability. Future work can concisely study programmability by combining quantitative and qualitative approaches. The study could consider other quantitative metrics such as the number of keywords, additional functions, and pointers necessary to port an application to GPUs. Concerning a qualitative approach, it could conduct experiments with undergraduate students. The students can implement parallel algorithms using GSParLib, SPar, and state-of-the-art GPU frameworks such as CUDA, OpenCL, and OpenACC. Then it is possible to collect how long it takes to develop the GPU code and collect other information. For instance, the students can report the main difficulties and facilities of using each framework.

- **Evaluate SPar's GPU extension on data parallelism applications.** We evaluated SPar's GPU extension only on stream processing benchmarks. When targeting stream benchmarks, SPar generates several routines to control the flow of data, and when using the GPU, SPar combines the CPU and GPU. However, SPar can also generate parallel code for data parallelism without generating routines concerning the stream processing domain. In this sense, future work can evaluate SPar's performance and programmability when using the GPU on applications that only offer data parallelism, such as the benchmarks in the NPB suite.

6.1 List of published papers

6.1.1 Publications as main author

- **Efficient NAS Benchmark Kernels with CUDA.** 28th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Västerås, 2020 [AGDF20]. This paper provides a CUDA implementation for the NPB kernels (CG, EP, FT, IS, and MG) and reports our experience programming the benchmarks.
- **NAS Parallel Benchmarks with CUDA and beyond.** Software: Practice and Experience (SPE), 2021 [AGR+21]. This paper continues our previous work. We provide a new CUDA implementation for the NPB kernels (CG, EP, FT, IS, and MG) and pseudo-applications (BT, LU, and SP). We define a set of design principles for GPU programming and use those principles to guide the implementations. We conduct a comparative study with the literature, analyzing parallelism strategies, programmability, and performance. Finally, we discuss how different aspects of algorithms can impact GPU performance and how they are related to GPU parameters.

6.1.2 Publications as co-author

- **The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures.** Future Generation Computer Systems (FGCS), 2021 [LGM+21]. This paper provides a port of the NPB from the original Fortran to C++. Additionally, we conduct a broad study of parallel programming for multi-core architectures by implementing the NPB with OpenMP, TBB, and FastFlow. We conducted experiments with different multi-core systems (Intel, IBM Power, and AMD) and compilers (GCC, ICC, and Clang).

REFERENCES

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

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

[AGR+21] Araujo, G.; Griebler, D.; Rockenbach, D. A.; Danelutto, M.; Fernandes, L. G. “NAS parallel benchmarks with CUDA and beyond”, *Software: Practice and Experience*, vol. In press, Nov 2021, pp. 1–28.

[AGT14] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of stream processing: application design, systems, and analytics”. Cambridge University Press, 2014, 558p.

[AMD] AMD Corporation. “HIP”. Source: https://github.com/ROCm-Developer-Tools/HIP, Visited on 2022/02/15.

[APD+14] Aldinucci, M.; Pezzi, G. P.; Drocco, M.; Tordini, F.; Kilpatrick, P.; Torquati, M. “Parallel video denoising on heterogeneous platforms”. In: Proceedings of the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems, 2014, pp. 1–8.

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

[ASA21] Altun, I.; Sahin, H.; Aslantas, M. “A new approach to fractals via best proximity point”, *Chaos, Solitons and Fractals*, vol. 146, May 2021, pp. 1–7.

[BBB+94] Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Simon, H. D.; Venkatakrishnan, V.; Weeratunga, S. K. “The NAS parallel benchmarks”, Technical Report, NASA Advanced Supercomputing Division, 1994, 79p.

[BFH+04] Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. “Brook for GPUs: stream computing on graphics hardware”, *ACM Transactions on Graphics*, vol. 23, Aug 2004, pp. 777–786.

[BHS+95] Bailey, D.; Harris, T.; Saphir, W.; Wijngaart, R. V. D.; Woo, A.; Yarrow, M. “The NAS parallel benchmarks 2.0”, Technical Report, NASA Advanced Supercomputing Division, 1995, 24p.

[CGM14] Cheng, J.; Grossman, M.; McKercher, T. “Professional CUDA C programming”. Wrox Press Ltd., 2014, 527p.

[CJ17] Chandrasekaran, S.; Juckeland, G. “OpenACC for programmers: concepts and strategies”. Addison-Wesley Professional, 2017, 320p.

[CLRS09] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. “Introduction to algorithms”. The MIT Press, 2009, 1292p.

[Coo13] Cook, S. “CUDA programming: a developer’s guide to parallel computing with GPUs”. Morgan Kaufmann Publishers Inc., 2013, 591p.

[DKO+19] Do, Y.; Kim, H.; Oh, P.; Park, D.; Lee, J. “SNU-NPB 2019: parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2019, pp. 93–105.

[DR13] Dummler, J.; Runger, G. “Execution schemes for the NPB-MZ benchmarks on hybrid architectures: a comparative study”. In: Proceedings of the Parallel Computing: Accelerating Computational Science and Engineering, 2013, pp. 733–742.

[DSU20] Daleiden, P.; Stefik, A.; Uesbeck, P. M. “GPU programming productivity in different abstraction paradigms: a randomized controlled trial comparing CUDA and thrust”, *ACM Transactions on Computing Education*, vol. 20, Oct 2020, pp. 1–27.

[ELK18] Ernstsson, A.; Li, L.; Kessler, C. “SkePU 2: flexible and type-safe skeleton programming for heterogeneous parallel systems”, *International Journal of Parallel Programming*, vol. 46, Feb 2018, pp. 62–80.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: a DSL for high-level and productive stream parallelism”, *Parallel Processing Letters*, vol. 27, Mar 2017, pp. 1–20.

[GHDF18a] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “High-level and productive stream parallelism for dedup, ferret, and bzip2”, *International Journal of Parallel Programming*, vol. 47, Feb 2018, pp. 253–271.

[GHDF18b] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream parallelism with ordered data constraints on multi-core systems”, *Journal of Supercomputing*, vol. 75, Jul 2018, pp. 4042–4061.

[GLM+18] Griebler, D.; Löff, J.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. “Efficient NAS benchmark kernels with C++ parallel programming”. In: Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2018, pp. 733–740.

[Gri16] Griebler, D. “Domain-specific language & support tool for high-level stream parallelism”, PhD’s Thesis, School of Technology - PPGCC - PUCRS, 2016, 243p.

[HA11] Han, T. D.; Abdelrahman, T. S. “hiCUDA: high-level GPGPU programming”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, Jan 2011, pp. 78–90.

[HGDF20] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream parallelism annotations for multi-core frameworks”. In: Proceedings of the Brazilian Symposium on Programming Languages, 2020, pp. 48–55.

[HLG22] Hoffmann, R.; Löff, J.; Griebler, D. “OpenMP as runtime for providing high-level stream parallelism on multi-cores”, *The Journal of Supercomputing*, vol. 78, Jan 2022, pp. 7655–7676.

[HLLL22] Huang, T.; Lin, D.; Lin, C.; Lin, Y. “Taskflow: a lightweight parallel and heterogeneous task graph computing system”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, Jun 2022, pp. 1303–1320.

[HMMT13] Hoshino, T.; Maruyama, N.; Matsuoka, S.; Takaki, R. “CUDA vs OpenACC: performance case studies with kernel benchmarks and a memory-bound CFD application”. In: Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 136–143.

[JFY99] Jin, H.; Frumkin, M.; Yan, J. “The OpenMP implementation of NAS parallel benchmarks and its performance”, Technical Report, NASA Advanced Supercomputing Division, 1999, 26p.

[KH10] Kirk, D. B.; Hwu, W. W. “Programming massively parallel processors: a hands-on approach”. Morgan Kaufmann Publishers Inc., 2010, 435p.

[KL20] Kang, P.; Lim, S. “A taste of scientific computing on the GPU-accelerated edge device”, *IEEE Access*, vol. 8, Nov 2020, pp. 337–347.

[KS19] Kuznetsov, E.; Stegailov, V. “Porting CUDA-based molecular dynamics algorithms to AMD ROCm platform using HIP framework: performance analysis”. In: Proceedings of the Supercomputing, 2019, pp. 121–130.

[KSCK19] Kimura, R.; Seigo, M.; Chipman, R. A.; Kitagawa, S. “Optical simulation for illumination using GPGPU ray tracing”. In: Proceedings of the Physics and Simulation of Optoelectronic Devices, 2019, pp. 171–179.

[Kum02] Kumar, V. “Introduction to parallel computing”. Addison-Wesley Longman Publishing Co., Inc., 2002, 856p.

[LGM+21] Löff, J.; Griebler, D.; Mencagli, G.; Araujo, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures”, *Future Generation Computer Systems*, vol. 125, Jul 2021, pp. 743–757.

[LHGF21] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level stream and data parallelism in C++ for multi-cores”. In: Proceedings of the Brazilian Symposium on Programming Languages, 2021, pp. 1–8.

[LTA+22] 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, Dec 2022, pp. 1–17.

[LTO+11] Lee, J.; Tran, M. T.; Odajima, T.; Boku, T.; Sato, M. “An extension of XcalableMP PGAs lanaguage for multi-node GPU clusters”. In: Proceedings of the Euro-Par: Parallel Processing Workshops, 2011, pp. 429–439.

[Lut] Lutz, K. “Boost.Compute”. Source: https://github.com/boostorg/compute, Visited on 2022/02/15.

[MGM+11] Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. “OpenCL programming guide”. Addison-Wesley Professional, 2011, 603p.

[MRR12] McCool, M.; Reinders, J.; Robison, A. “Structured parallel programming: patterns for efficient computation”. Morgan Kaufmann Publishers Inc., 2012, 432p.

[MSM04] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for parallel programming”. Addison-Wesley Professional, 2004, 384p.

[MWUP20] Morrical, N.; Wald, I.; Usher, W.; Pascucci, V. “Accelerating unstructured mesh point location with RT cores”, *IEEE Transactions on Visualization and Computer Graphics*, vol. In press, Dec 2020, pp. 1–14.

[NAS] NASA Advanced Supercomputing Division. “NAS parallel benchmarks”. Source: https://www.nas.nasa.gov/publications/npb.html, Visited on 2022/02/15.

[NHJ21] Niu, X.; He, H.; Jin, M. “Application of ray-tracing method in electromagnetic numerical simulation algorithm”. In: Proceedings of the International Applied Computational Electromagnetics Society Symposium, 2021, pp. 1–2.

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

[NVI] NVIDIA Corporation. “Thrust quick start guide”. Source: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf, Visited on 2022/02/15.

[NVI12] NVIDIA Corporation. “Optimizing parallel reduction in CUDA”, Technical Report, NVIDIA Corporation, 2012, 38p.

[NVI20a] NVIDIA Corporation. “CUDA C programming guide”, Technical Report, NVIDIA Corporation, 2020, 379p.

[NVI20b] NVIDIA Corporation. “NVIDIA A100 tensor core GPU architecture”, Technical Report, NVIDIA Corporation, 2020, 82p.

[NVI22] NVIDIA Corporation. “GPU-accelerated applications”, Technical Report, NVIDIA Corporation, 2022, 88p.

[Opea] OpenACC Organization. “The OpenACC application programming interface”. Source: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf, Visited on 2022/02/15.

[Opeb] OpenMP Organization. “OpenMP application program interface version 5.0”. Source: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, Visited on 2022/02/15.

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

[Pie20] Pieper, R. L. “High-level programming abstractions for distributed stream processing”, Master’s Thesis, School of Technology - PPGCC - PUCRS, 2020, 170p.

[Roc20] Rockenbach, D. A. “High-level programming abstractions for stream parallelism on GPUs”, Master’s Thesis, School of Technology - PPGCC - PUCRS, 2020, 163p.

[RSG+19] Rockenbach, D. A.; Stein, C. M.; Griebler, D.; Mencagli, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “Stream processing on multi-cores with GPUs:

parallel programming models' challenges". In: Proceedings of the International Parallel and Distributed Processing Symposium Workshops, 2019, pp. 834–841.

[SE15] Stone, C. P.; Elton, B. H. "Accelerating the multi-zone scalar pentadiagonal CFD algorithm with OpenACC". In: Proceedings of the Workshop on Accelerator Programming Using Directives, 2015, pp. 1–7.

[SGDF19] Stein, C. M.; Griebler, D.; Danelutto, M.; Fernandes, L. G. "Stream parallelism on the LZSS data compression application for multi-cores with GPUs". In: Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2019, pp. 247–251.

[SJL11] Seo, S.; Jo, G.; Lee, J. "Performance characterization of the NAS parallel benchmarks in OpenCL". In: Proceedings of the IEEE International Symposium on Workload Characterization, 2011, pp. 137–148.

[SK10] Sanders, J.; Kandrot, E. "CUDA by example: an introduction to general-purpose GPU programming". Addison-Wesley Professional, 2010, 311p.

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

[SRG+20] Stein, C. M.; Rockenbach, D. A.; Griebler, D.; Torquati, M.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. "Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units", *Concurrency and Computation: Practice and Experience*, vol. 33, May 2020, pp. 1–19.

[STKC17] Shen, C.; Tian, X.; Khaldi, D.; Chapman, B. "Assessing one-to-one parallelism levels mapping for OpenMP offloading to GPUs". In: Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores, 2017, pp. 68–73.

[TLGA+22] Trott, C. R.; Lebrun-Grandié, 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, Apr 2022, pp. 805–817.

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

[VGF21] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing high-level self-adaptive abstractions for stream parallelism on multi-cores”, *Software: Practice and Experience*, vol. 51, Jan 2021, pp. 1194–1217.

[VRJ+20] Vogel, A.; Rista, C.; Justo, G.; Ewald, E.; Griebler, D.; Mencagli, G.; Fernandes, L. G. “Parallel stream processing with MPI for video analytics and data visualization”. In: Proceedings of the High Performance Computing Systems, 2020, pp. 102–116.

[WF02] Wijngaart, R. V. D.; Frumkin, M. “NAS grid benchmarks version 1.0”, Technical Report, NASA Advanced Supercomputing Division, 2002, 18p.

[Whe] Wheeler, D. A. “SLOCCount”. Source: https://dwheeler.com/sloccount/, Visited on 2022/02/15.

[WJ03] Wijngaart, R. V. D.; Jin, H. “The NAS parallel benchmarks, multi-zone versions”, Technical Report, NASA Advanced Supercomputing Division, 2003, 9p.

[WUM+19] Wald, I.; Usher, W.; Morrical, N.; Lediaev, L.; Pascucci, V. “RTX beyond ray tracing: exploring the use of hardware ray tracing cores for tet-mesh point location”. In: Proceedings of the High-Performance Graphics, 2019, pp. 1–7.

[Xca] XcalableMP Specification Working Group. “XcalableMP language specification version 1.4”. Source: https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf, Visited on 2022/02/15.

[XTC+14] Xu, R.; Tian, X.; Chandrasekaran, S.; Yan, Y.; Chapman, B. “NAS parallel benchmarks for GPGPUs using a directive-based programming model”. In: Proceedings of the Languages and Compilers for Parallel Computing, 2014, pp. 67–81.

[YTZ21] Yu, D.; Ta, W.; Zhou, Y. “Fractal diffusion patterns of periodic points in the mandelbrot set”, *Chaos, Solitons and Fractals*, vol. 153, Dec 2021, pp. 1–9.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br