High-Level Stream and Data Parallelism in C++ for GPUs

Dinei A. Rockenbach
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
dinei.rockenbach@edu.pucrs.br

Júnior Löff
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
junior.loff@edu.pucrs.br

Gabriell Araujo
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
gabriell.araujo@edu.pucrs.br

Dalvan Griebler
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
dalvan.griebler@pucrs.br

Luiz Gustavo Fernandes
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
luiz.fernandes@pucrs.br

ABSTRACT

GPUs are massively parallel processors that allow solving problems that are not viable to traditional processors like CPUs. However, implementing applications for GPUs is challenging to programmers as it requires parallel programming to efficiently exploit the GPU resources. In this sense, parallel programming abstractions, notably domain-specific languages, are fundamental for improving programmability. SPar is a high-level Domain-Specific Language (DSL) that allows expressing stream and data parallelism in the serial code through annotations using C++ attributes. This work elaborates on a methodology and tool for GPU code generation by introducing new attributes to SPar language and transformation rules to SPar compiler. These new contributions, besides the gains in simplicity and code reduction compared to CUDA and OpenCL, enabled SPar achieve 331% of higher throughput when exploring combined CPU and GPU parallelism, and 665% when using batching.

KEYWORDS

Programming language, parallelism, parallel patterns, algorithmic skeletons, C++ annotations, source-to-source code generation

ACM Reference Format:
Dinei A. Rockenbach, Júnior Löff, Gabriell Araujo, Dalvan Griebler, and Luiz Gustavo Fernandes. 2022. High-Level Stream and Data Parallelism in C++ for GPUs. In *XXVI Brazilian Symposium on Programming Languages (SBLP 2022), October 6–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3561320.3561327

1 INTRODUCTION

Stream processing systems have been increasing in popularity over the last few years. They are especially relevant for dealing with the large amount of data being produced by live sources such as IoT devices, social media, and financial markets. However, in order to achieve efficient computation, programmers need to carefully accommodate the streaming application according to the underlying computing resources. This task usually requires parallelism strategies and other low-level optimizations, which are challenging for most application programmers. Usually when programmers try to implement efficient code, they end up mixing the application business logic with the parallelism strategy. Consequently, the code quickly becomes complex, and ordinary activities such as implementing, debugging, and maintaining code become cumbersome and error-prone tasks. Furthermore, modern applications can benefit from the composition of different parallelism strategies to improve performance. For example, stream processing applications usually expose data parallelism within some streaming stage.

In this sense, new methodologies promote parallel programming abstractions to ease the task of writing parallel code. Commonly, we organize parallel programming abstractions into three layers or levels: 1) a set of fundamental and low-level mechanisms that allow accessing hardware features such as triggering or synchronizing threads (e.g., CUDA); 2) a set of parallel patterns that hide many lower-level complexities via templates that work as ready-to-use parallelism strategies (e.g., Intel TBB); and 3) domain-specific languages that employ high-level abstractions via code annotations that are used to generate automatic parallel code (e.g., SPar).

The state-of-the-art parallel programming abstractions targeting general purpose CPUs are mostly from the second-level. The programmer is equipped with composable, parametric, and reusable abstractions that can be inter-connected to help modeling complex data streams. Apart from that, GPUs are powerful architectures that are equipped with thousands of cores targeting problems that are complex to be solved on CPUs. When the programmer needs to deal with specialized computing systems like GPUs, the available solutions he can use are essentially from the first-level of parallelism abstractions. Therefore, programmers need to reason about a new lower-level language and design a new strategy for combining multi-core and many-core parallelism.

When we inspect the available frameworks for GPU programming in the literature, the standard tools are CUDA and OpenCL.

Also, there are some alternatives in the industry and scientific community. Frameworks like HIP [2] offer wrappers over CUDA and OpenCL, which facilitates developing portable code between GPUs of different vendors. Some frameworks like FastFlow, Kokkos, SkePU, and Thrust additionally offer parallel patterns via structured parallel programming to approach GPUs. Other frameworks such as OpenACC, OpenMP, and hiCUDA provide code annotations do abstract some complexities, but they still require hardware knowledge and specific optimizations. Despite the differences between GPU supported frameworks, all of them share a common characteristic: they require significant programming efforts and knowledge about parallelism and hardware aspects. In fact, very few frameworks tried to provide abstractions for stream parallelism targeting CPUs and GPUs simultaneously.

Considering the literature limitations, in this paper, we leverage SPar's high-level language and extend it to support parallel code for CPUs combined with GPUs on stream processing applications. SPar is a Domain Specific Language (DSL) embedded in C++ that offers third-level abstractions to express stream parallelism via code annotations. However, SPar was only generating code for multi-core architectures. In this work we define and implement new transformation rules in SPar's compiler to make it able to generate automatic parallel code for CPUs combined to GPUs via source-to-source code transformations. Our main scientific contributions are: (1) a high-level language extension to support GPUs; (2) a code generation methodology for the compiler that translates high-level annotations to parallel code; and (3) an evaluation of our proposed methodology using three applications from different domains.

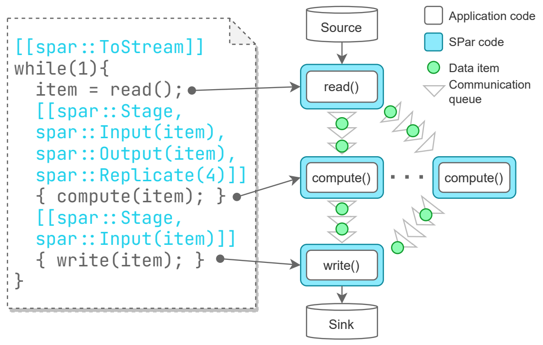
The remainder of this paper is organized as follows. Section 2 gives a bird's eye view of SPar, highlights our motivations, and introduces our new language for GPUs along with the compiler methodology for source-to-source parallel code generation. Section 3 presents the results of our experiments. Section 4 introduces our related work and Section 5 the conclusion and future work.

2 HIGH-LEVEL STREAM AND DATA PARALLELISM

In this section, we introduce a programming model for expressing stream and data parallelism in stream processing applications targeting multi-cores and GPUs. The outline of this section is the following: Section 2.1 introduces the SPar programming model. Section 2.2 describes our motivation for extending SPar to offload streaming data-intensive computation routines into GPUs. Section 2.3 introduces the new high-level language that enables stream and data parallelism annotations in C++ code. Then, in Section 2.4 we leverage the new language and implement a new compiler methodology that automatically generates heterogeneous parallel code using source-to-source code transformations.

2.1 SPar Programming Model

SPar (acronym for Stream Parallelism) is a programming model for expressing stream parallelism in C++ codes. SPar was first introduced in 2016 [8, 10] and is being built upon since. Currently, different research works are being conducted in SPar's ecosystem, mainly they target different architectures (i.e. CPUs, Clusters, and



```
[[spar::ToStream]]
while(1){
  item = read();
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item),
  spar::Replicate(4)]]
  { compute(item); }
  [[spar::Stage,
  spar::Input(item)]]
  { write(item); }
}
```

Figure 1: SPar annotations and the data flow generation.

low-resource hardware devices) and extend support to further parallelism paradigms (i.e. data parallelism and data flow).

SPar aims at providing higher levels of abstraction to hide the difficulties inherited of computer architectures and systems and the challenge of writing parallel code. SPar design principles are towards productivity and portability. It provides a clear separation of concerns between the application business logic and parallelism details. Therefore, programmers that employ SPar can focus on the application, while SPar's compiler is in charge of providing parallelism-specific optimizations.

SPar equips programmers with a domain-specific language (DSL) that can be used for annotating data stream regions in sequential C++ code. Afterwards, the SPar compiler analyses these information and automatically generates parallel code. The code generation is accomplished using source-to-source transformations performed directly in the standard C++ AST (abstract syntax tree). Since the SPar compiler represents the full semantics of the C++ standard in an internal AST, it gives SPar the support required to perform complex and powerful code transformations.

The SPar language was initially conceptualized via five domain-specific attributes: (1) `ToStream` denotes the scope of a data stream in the code (can be a loop constantly receiving data); (2) `Stage` denotes the scope of a sequential stage/block (can be a computation step applied to each item of the data flow); (3 and 4) `Input` and `Output`, as the name suggests, are the inputs and outputs of a data stream region or a processing stage; (5) `Replicate` is a special attribute that informs a processing stage could be replicated.

The programmer may use the aforementioned attributes in order to express information about the data stream in a sequential C++ code. Note that the programmer uses a high-level language and does not deal with low-level code. For instance, Figure 1 shows a traditional stream processing application annotated with SPar. The application constantly reads data from a source, applies a computation step and writes results into a sink. The `ToStream` indicates the annotated region of code represents a data flow. In this region, each item read from a source is processed by two `Stages`. The first `Stage` consumes a data item, applies a computational routine over the data and sends it forward. Also, the `Replicate` attribute indicates the stage is replicated using the specified number. The second `Stage` consumes the previous data item and writes them into a sink.

A high-level representation of the parallelism mechanisms generated by SPar is represented in cyan on the right-hand side of

Figure 1. After analyzing the information provided by the programmer, SPar's compiler converges into a suitable parallel data stream by employing the Farm pattern. Once the parallel pattern was selected, SPar can generate the parallel code targeting different runtimes. The original SPar version [9] generates parallel code using the FastFlow library. Recent works [14, 15] extended SPar to support parallel code generation targeting Intel Threading building blocks and OpenMP.

2.2 Motivation for language extension

In the last few years, the rise of massively parallel hardware and the performance differences of the multi-core and many-core architectures led developers to move computationally intensive (parallel) parts of programs to accelerators (such as GPUs). However, programming for many-core hardware poses additional challenges concerning parallel programming for multi-core machines, due to the differences in the architectural design and separate memory spaces. It is a challenge to synchronize computation and data between different computing systems. Usually, programmers require to design and implement exclusive low-level parallelism strategies particular to each application and computing system architecture.

Modern High-Performance Computing (HPC) servers are composed of a combination of multi-core CPUs and many-core GPUs. In order to take advantage and efficiently exploit the underlying parallel resources, applications programmers rely on available APIs (Application Programming Interface) or parallel programming models. While many tools for GPU programming does not offer efficient abstractions for stream parallelism [6, 31], other tools are not able to efficiently exploit the computing resources since they do not consider the data parallelism exposed by the stream processing applications [16, 32]. Even tools that support stream and data parallelism require significant code refactoring in order to exploit the heterogeneous hardware [1].

Ideally, the tools developed by system programmers should provide efficient abstractions that does not require stream processing application programmers to learn hardware details in order to exploit the parallelism available in the computer architecture. However, the lack of high-level abstractions to explore these architectures was limiting SPar's usage among application programmers. Although possible, exploiting GPU parallelism using SPar annotations required much effort and deep knowledge about the underlying architecture [29]. The current SPar attributes are closely related to the stream parallelism domain. Also, they do not express any semantics of the data parallelism properties. It was necessary an extension to SPar language to express data parallelism along with stream parallelism.

Therefore, we posed ourselves the challenge to design efficient and high-level parallel programming abstractions for expressing parallelism on stream processing applications targeting heterogeneous parallel computer architectures, without substantial changes to the original syntax and semantics. We propose a simple and expressive unified programming model for expressing stream and data parallelism using C++ attributes. We introduce the new language and compiler strategies in the following sections.

2.3 Data parallelism attributes in SPar

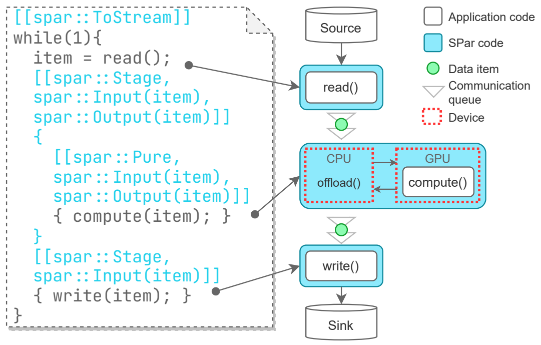
In order to safely generate parallel code, the compiler must be sure that the operation being applied to the data elements can be executed in parallel, i.e. it is a *pure* function: "whose output depends only on its input and does not modify any other system state" [21]. Functional programming semantics defines a pure function as "a function that, given the same input, will always return the same output and does not have any observable side effect" [19]. Since there is no standard way of automatically detecting this property in a given C++ code block [3, 7, 26], the application programmer must provide this information. In the following we present the SPar language extension we propose to support data parallelism:

2.3.1 Pure. None of the current SPar attributes (presented in Section 2.1) carries information about the *pureness* of the code. Thus, we created a novel attribute called `Pure` to identify operations that can be safely executed in parallel [28]. The `Pure` attribute indicates that the annotated code block is a pure function. This attribute may be used along with the `Stage` attribute list to mark the entire `Stage` as pure, or as an identifier attribute inside code regions annotated with `Stage` to mark specific portions of the `Stage` region as pure operations. The input and output data of the pure region are defined by the `Input` and `Output` attributes. In SPar, a `Stage` or code block is considered a pure function when it satisfies the following statements to guarantee correct use and correct code generation:

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

From the programmer perspective, the `Pure` is another attribute that increases the language expressiveness. It enables programmers to identify and annotate data parallelism inside the `Stage`. On the other hand, the compiler transformation rule identifies that this region/function can be computed in parallel over multiple data. It is up to the compiler scan the available hardware and decide to which parallel architecture (GPU or multi-core) generate the stream parallelism with data parallelism code. Figure 2 presents a high-level representation of the transformations performed by SPar's compiler in the presence of the `Pure` attribute. Besides generating the code for the stream management, as it was illustrated in Figure 1, SPar generates the code for host-device data transfer and communication (represented as `offload()` in Figure 2) and invokes the pure function (`compute()`) in the accelerator.

Figure 3 presents examples of more complex annotation schemas. Figure 3(a) shows a `Pure` region calling `compute_A()` inside a `Stage` that is not replicated and then a replicated `Stage` calling `compute_B()`. The activity graph shows that SPar generates the GPU code and then sends the outputted data items to the `n` replicated workers. Figure 3(b) shows a `Pure` attribute being used inside a replicated `Stage`. In this case, SPar leverages the thread-safety capabilities of the underlying runtime library to manage multiple workers invoking kernels on the GPU simultaneously.



```
[[spar::ToStream]]
while(1){
  item = read();
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item)]]
  {
    [[spar::Pure,
    spar::Input(item),
    spar::Output(item)]]
    { compute(item); }
  }
  [[spar::Stage,
  spar::Input(item)]]
  { write(item); }
}
```

Figure 2: Example of SPar annotations for GPU offloading.

2.3.2 Impure. As explained before, the Pure attribute enables programmers to express stateless data parallelism. However, pure regions can perform stateful or "impure" operations that hinders the ability to exploit parallelism in this region. For instance, a single line of code with side effects, by definition, would classify the entire block of code as being not pure. For enabling SPar to leverage the entire properties of a Pure block of code, users would require to manually deal and synchronize the impure operations while "purifying" them. Alternatively, we equipped programmers with the Impure attribute to identify impure regions inside pure blocks. Therefore, programmers can use the *Impure* to annotate the code region they want to "purify". Then, SPar's compiler will try to automatically implement the required synchronization mechanisms to allow parallelism. For example, when targeting multi-cores an impure region of code is purified using locks or other optimizations such as the reduce parallel pattern [20]. In this work, we already implemented an optimization that identify Reduce patterns and automatically generate the required synchronization between parallel GPU threads and kernels. Other abstractions for the Impure attribute that can be investigated in the future are speculative synchronization mechanisms, different parallel patterns, and GPU atomic operations.

2.3.3 Data Management. When designing our high-level language, we also identified that in order to automatically manage data copies between host and device memories, the SPar compiler must know the length of any data to be copied. This also applies to other computing architectures supported by SPar such as distributed memory. Therefore, in our language we propose a modified syntax to express vector and array sizes in the SPar Input and Output attributes. We expect the programmer to annotate the size of a contiguous allocated memory space via Input(data[size]) or output(data[size]). The data can be statically or dynamically allocated, and can be of any data type, however, the declaration of custom types must be accessible by the compiler.

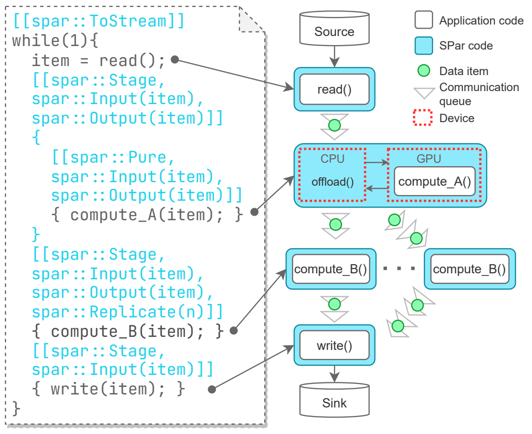
2.3.4 Batch. A previous work [29] has evaluated different parallel programming models when combining stream and data parallelism. One of their conclusions is that fine-grained stream processing may not generate enough workload to properly exploit massively parallel architectures such as GPUs. Thus, some stream processing applications may not provide the expected performance scalability when using GPUs. For these cases, we are providing the possibility to express stream batches in SPar through the new auxiliary attribute for the Stage, named Batch, which activates the batching optimization for this specific Stage [13]. The programmer can specify as argument the size of the batch with literal or integer variable. In principle, this is the amount of stream items to be computed at once by the annotated stage, which must be a Pure stage. In short, Batch allows programmers to define the stream item granularity. Figure 4 represents the transformations performed by SPar's compiler when the attributes Pure and Batch are used together with a Stage attribute. SPar generates the code to accumulate N (which is 4 in Figure 4) data items and processes them together in the GPU. Data source and sink are omitted in the Figure for simplicity.

2.4 Parallel Code Generation

The SPar programming model is based on the C++ attributes presented in Section 2.1, which we extended by adding the attributes presented in Section 2.3. Figure 5 presents an overview of SPar's methodology for parallel code generation. The attributes defined in the SPar language are combined in annotation schemas, following the definitions to ensure correct usage. The rules define parallel patterns that are generated based on each annotation schema and are implemented in the compiler.

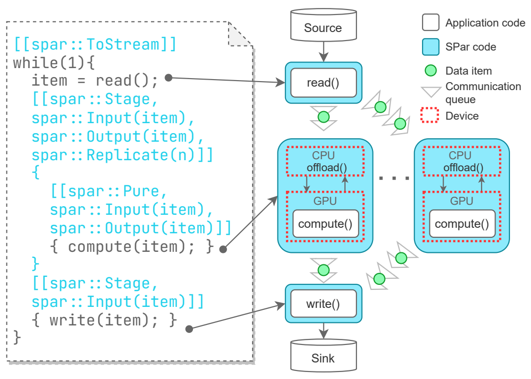
The compiler implementation follows a three-step approach: (1) the compiler scans the code and parses the C++ syntax, validating the combination of attributes in annotation schemas. After parsing the code the compiler generates an Abstract Syntax Tree (AST), which is then analyzed to extract information from the annotated source code. The analysis of the AST identifies the attributes being used, optimization opportunities, and extract any information needed for the next steps; (2) the compiler matches the transformation rules defined in the language to the annotated code, deciding which parallel pattern will be generated according to the annotation schema. This is performed in a per-annotation schema basis, i.e., the transformation rules implemented in the compiler are checked individually against each annotation schema in the code, which allows programmers to apply many annotation schemas in a single source file. By checking all transformation rules for each annotation schema, the compiler also allows for both data and stream parallelism to be exploited in a single annotation schema; and (3) the compiler then applies the transformation rules and transforms the AST code by inserting calls to lower-level runtime libraries that implement the parallel patterns. The transformed AST is then converted into C++ code and compiled into a binary executable.

The original rules to generate the Pipeline and Farm parallel patterns were presented in [9]. An example of the Farm pattern generated from an annotation schema is presented in Figure 1. With the novel attributes for data parallelism, we extended the existing rules to generate the data-parallel patterns Map, Reduce, and MapReduce. The full set of attributes now allow transformation rules targeting stream and data parallelism to be combined. Figures 2 and 3 present examples of combined stream and data parallelism being generated by the SPar compiler targeting an heterogeneous architecture. We implemented the data parallel patterns for both CUDA and OpenCL via C++ template library in order to



```
[[spar::ToStream]]
while(1){
  item = read();
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item)]]
  {
    [[spar::Pure,
    spar::Input(item),
    spar::Output(item)]]
    { compute_A(item); }
  }
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item),
  spar::Replicate(n)]]
  { compute_B(item); }
  [[spar::Stage,
  spar::Input(item)]]
  { write(item); }
}
```

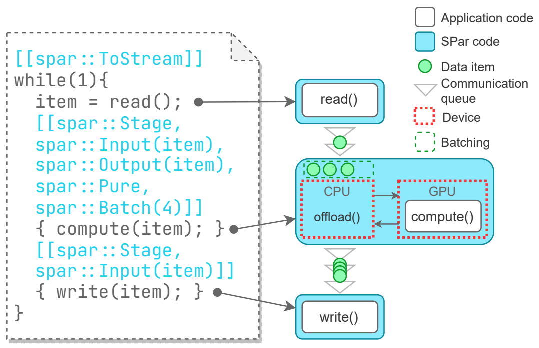
(a) Pure region before Replicated Stage.



```
[[spar::ToStream]]
while(1){
  item = read();
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item),
  spar::Replicate(n)]]
  {
    [[spar::Pure,
    spar::Input(item),
    spar::Output(item)]]
    { compute(item); }
  }
  [[spar::Stage,
  spar::Input(item)]]
  { write(item); }
}
```

(b) Pure region inside Replicated Stage.

Figure 3: Example of SPar annotations for GPU offloading in complex graphs.



```
[[spar::ToStream]]
while(1){
  item = read();
  [[spar::Stage,
  spar::Input(item),
  spar::Output(item),
  spar::Pure,
  spar::Batch(4)]]
  { compute(item); }
  [[spar::Stage,
  spar::Input(item)]]
  { write(item); }
}
```

Figure 4: Representation of SPar's batching implementation.

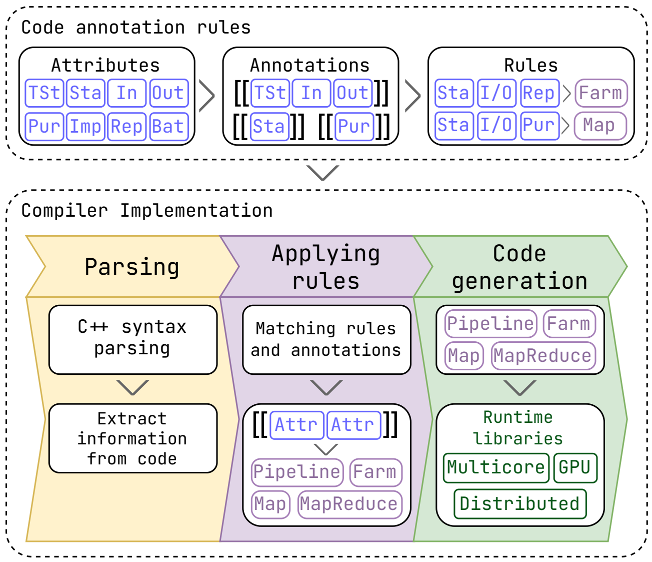


Figure 5: Methodology for Parallel Code Generation.

avoid vendor lock-in (enabling different GPU vendors). This library is an intermediate code representation of low-level template-based parallelism implementations when generating GPU parallel code.

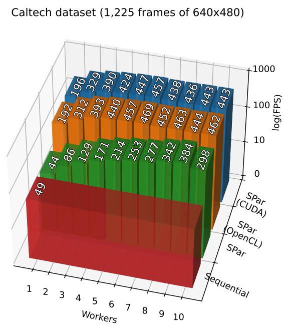
2.4.1 Automatic and Semi-automatic Optimizations. When combining stream and data parallelism, the compiler inserts the code for preparing and compiling the GPU kernel before the streaming region so that these initialization steps are performed only once for the entire processing of the stream. Some of the objects of CUDA and OpenCL runtimes cannot be shared among the host threads used to exploit stream parallelism. During the code generation, the compiler handles them appropriately when generating the GPU parallel code.

If the Batch attribute is present in a Stage annotation, the compiler adds a vector of stream items in the stage structure to store all the incoming stream items. When the vector reaches the size defined as argument of the Batch attribute or if the stream comes to an end, the entire batch of items is processed at once in a single GPU kernel invocation. This process increases the latency but improves throughput in cases where stream items do not expose enough computation to be worth offloading to the GPU. The application developer should consider this trade-off between throughput and latency to decide whether to use the Batch attribute and the batch size that best suits their needs. We present details of this trade-off in Section 3.3.

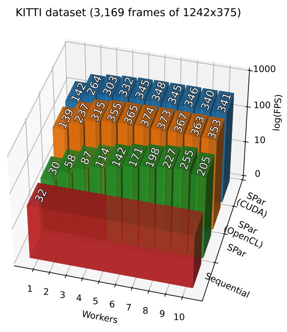
3 EXPERIMENTS

3.1 Methodology and Environment

The experiments were conducted on a machine equipped with a processor Intel i9-7900X @ 3.3 GHz (10 cores and 20 threads), 48 GB of RAM (3×16 GB DDR4 @ 2400 MT/s), and a GPU NVIDIA Titan Xp (3840 CUDA cores) with compute capability 6.1 and 12 GB GDDR5X @ 2400 MHz of memory. The operating system was Ubuntu 20.04 LTS (kernel 5.4.0-86-generic). The NVIDIA driver installed was the 450.102.04. The software utilized was CUDA Toolkit v11.0, OpenCL 1.2, and GCC 9.3 with -O3 compiler flag.



(a) Workload 1 (1225 frames).



(b) Workload 2 (3169 frames).

Figure 6: Results of Lane Detection (LD) throughput.

Each version of the benchmarks is named as follows: Serial version as `serial`. SPar parallel code for CPU as `SPar`. SPar parallel code for GPU combined with CPU as `SPar (CUDA)` (CUDA code generation) and `SPar (OpenCL)` (OpenCL code generation). The metrics were collected from ten executions of each test, a negligible standard deviation was observed in all tests.

3.2 Overall Performance Evaluation

In this section we present the throughput performance for three stream processing applications: Lane Detection (LD) [34], Mandelbrot Streaming (MB) [33], and Raytracer (RT) [17]. In the graphs, the X axis indicates the number of replicas of each stage, the Y axis lists the versions of the benchmarks, and Z axis presents the amount of throughput achieved, which are measured in Frames Per Second (FPS) in LD and RT, and Lines per second (Lines/s) in the MB. We tested the applications using two different workloads, where workload 1 has inputs with medium size and workload 2 has larger input sizes.

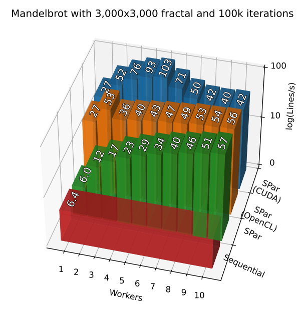
3.2.1 LD benchmark. Figure 6 shows the results of the LD benchmark. As can be seen, the GPU versions were up to 22% better than the best result of the CPU parallel version in Figure 6(a), and 47% in Figure 6(b). The difference of performance when varying the Workloads for the GPU occurs because the Workload 1 is a low resolution video. Processing small frames is not computationally intensive for GPUs, imposing a GPU under-utilization. The frames in the Workload 2 are larger, which improves the GPU performance. Using larger workloads on GPUs improves the performance mainly because it prevents the GPU cores from becoming idle, additionally it triggers schedule mechanisms for the GPUs that lower the latency of instructions and memory accesses. In the CPU parallel version we observe a more noticeable performance improvement when varying the number of workers, the same does not repeat in the GPU versions. Since the most intensive computational routines are performed by the GPU, CPU cores compute only a small amount of work and become idle. Finally, the performance of the CUDA and OpenCL code generated was similar.

3.2.2 MB benchmark. The results of MB benchmark are illustrated in Figure 7. The GPU versions achieved up to 81% better throughput than the best result of the CPU parallel version in Figure 7(a), and 135% in Figure 7(b). However, this performance improvement is only observed when SPar generates CUDA code, it occurs due to CUDA low-level mechanisms that are more optimized than OpenCL for NVIDIA GPUs. Different from LD, we observe a more noticeable performance difference when varying the amount of workers in the GPU versions. It occurs because the GPU usage is lower, and the CPU is responsible for computing more routines. This explains the better scaling when adding more parallel workers. However, the performance starts to decrease when using 6 workers or more as upon increasing the amount of workers also increases the overhead of communication in the application.

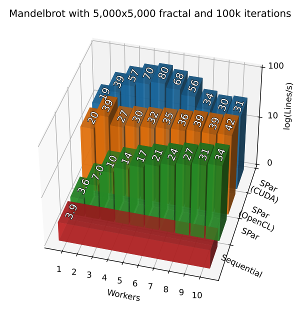
3.2.3 RT benchmark. Figure 8 shows the results of RT benchmark. The GPU versions were up to 286% better than the best result of the CPU parallel version in Figure 8(a), and 331% in Figure 8(b). Different from LD and MB, no performance improvement is observed when varying the amount of workers beyond 2. The GPU performs most of the computations and the application imposes a very little usage of the CPU, consequently, increasing the amount of workers does not provide major improvements.

3.3 Batching Evaluation

In this section we evaluate the impact of the batching feature we added to SPar. We chose to discuss only the MB SPar CUDA version for the sake of space, but our tests suggest that the same conclusions hold true to the OpenCL version and to the other Workloads as well.

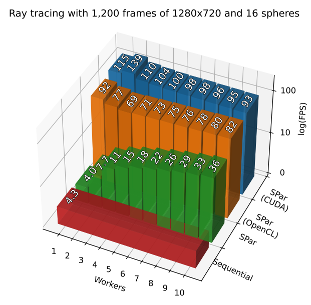


(a) Workload 1 (3000x3000 matrix).

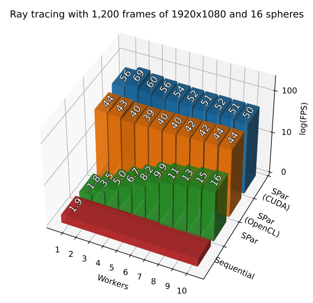


(b) Workload 2 (5000x5000 matrix).

Figure 7: Results of Mandelbrot Streaming (MB) throughput.

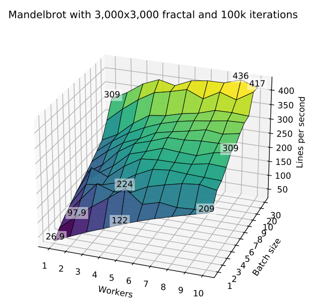


(a) Workload 1 (1280x720 resolution).

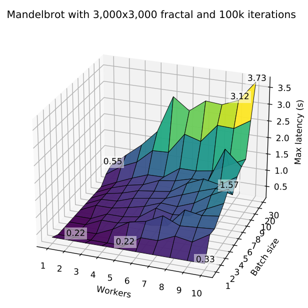


(b) Workload 2 (1920x1080 resolution).

Figure 8: Results of Raytracer (RT) throughput.



(a) Throughput.



(b) Latency.

Figure 9: SPar (CUDA) batching in Mandelbrot Streaming.

We measure the latency by adding a timestamp to each stream item in the first stage and checking the time it spent in the streaming until arriving in the last stage. We enabled an on-demand scheduling that only sends items when workers are ready to immediately start computing. This avoids measuring the time that the item spend waiting in the queue.

Usually, increasing the batch size also increases the latency since the items have to wait until the stages receive enough items to compute the batch. However, many stream processing applications have strict requirements over latency, defined as service level objectives (SLO) thresholds [11]. Therefore, it is desirable to maximize the throughput while keeping the latency within acceptable levels [30]. Figure 9 presents the impact of different numbers of workers and batch sizes in the MB SPar CUDA version. The X axis indicates the number of workers on each stage, the Y axis indicates the batch size, and the Z axis presents the amount of throughput (9(a)) or the maximum latency measured for any stream item (9(b)). We tested the application without batching (which is presented in Figure 9 as batch size of 1) and using batch sizes of 2 to 10, and also 20 and 30.

Unexpectedly, we found that a batch size of 2 actually reduces the latency in our experimental scenario. The reason is that the bottleneck is the GPU communication. Thus, the workers of the replicated stage must wait for the GPU response and the first stage waits for it to deliver the next item. This item is spending time waiting in the worker's queue. A batch size of 2 reduces the latency because the items are generated by the first stage faster than the replicated stages can process, so the worker of the replicated stage does not have to wait to receive the next item to compute the batch. In this case, the waiting time of the first item of each batch in the worker's batch vector is lower than the waiting time of the item in the worker's queue. This trend continues up to a batch size of 10 for 1 worker and up to a batch size of 5 for 2 workers, on which higher batch sizes start to increase the latency times. Therefore, the lowest maximum latency is 218 ms (0.22 s in Figure 9(b)), using 2 workers and a batch size of 2.

Besides reducing the latency, using a batch size of 2 offers an 86% of increase in the throughput: 97.9 lines/s using 2 workers and a batch size of 2 with respect to 52.5 lines/s using the same 2 workers but without batching. For this workload, the configuration of 5 workers with a batch size of 2 is also particularly interesting: it presents a throughput of 224 lines/s (84% more than the same configuration without batching, 122 lines/s) and the highest latency is only 220 ms (0.22 s in Figure 9(b)). While the MB GPU versions without batching (Figure 7(a)) were up to 80% better than the best result of the CPU parallel version, the GPU version with batching presented up to 665% of throughput improvement (436 lines/s using 9 workers and a batch size of 30 with respect to 57 lines/s of the CPU parallel version). When using batching, a single GPU kernel computes several elements at once, which impacts two main factors that improve the throughput: (1) it decreases the amount of GPU kernel launches and consequently the required communication between the CPU and GPU; and (2) it increases the workload size by merging several stream elements into a single GPU kernel invocation, improving the GPU usage. Increasing the number of workers or the batch size impacts the latency more significantly because the items are waiting longer in the worker's batch vector. The best batch size depends on the workload characteristics, but also in the latency and throughput requirements of the application.

Table 1: Source Lines of Code (SLOC) comparison.

App	Seq.	FF+CUDA	FF+OpenCL	SPar
MB	48	169 (+252%)	292 (+508%)	56 (+17%)
RT	455	595 (+31%)	789 (+73%)	463 (+2%)
LD	300	504 (+68%)	657 (+119%)	312 (+4%)

3.4 Programmability Considerations

In this Section we briefly discuss the programability aspects regarding SPar's programming model compared to manual implementations targeting CPU+GPU. We discuss programming characteristics of the versions and analyze the physical Source Lines of Code (SLOC) added to the sequential version. Table 1 summarizes the results. Although SLOC alone cannot be used to decide which interface is better, it can give a general idea of extra code lines that programmers need to design.

Table 1 shows that MB is the smallest sequential application in terms of lines of code. To enable CPU+GPU parallelism, the handwritten versions require almost 2.5x and 5x more lines of code for CUDA and OpenCL, respectively. SPar only required 8 extra lines of annotations, roughly 17%. Both CUDA and OpenCL present a verbose API, where programmers need to implement the mechanisms for data management, thread synchronization, and workload balancing. Instead, SPar equips programmers with a high-level and intuitive language. Programmers can use SPar's attributes to provide information about the data flow, while the compiler is in charge of data management, and other computational aspects like synchronization and schedulers. For RT and LD we observe similar behavior, where OpenCL requires the biggest amount of code lines, followed by CUDA. Again, SPar only requires 2% and 4% more lines of code compared to the sequential counterpart.

These results composed with the previous performance analysis from Section 3.2 indicates that our programming abstractions are efficient and suitable for stream processing applications. The high-level language we designed shows the lowest amount of code lines while the parallel code generated by our compiler achieves important throughput values.

Table 2: Comparison with related work.

Work	API	GPU Runtime	Supported Architectures	Parallelism Exploitation
OpenMP [25]	`pragma`	Compiler-dependent	GPUs and multi-cores	Low-level parallelism
OpenACC [23]	`pragma`	Compiler-dependent	GPUs	Low-level parallelism
hiCUDA [12]	`pragma`	CUDA	GPUs	Low-level parallelism
XMP-ACC [18]	`pragma`	CUDA	Distributed	Low-level parallelism
XACC [22]	`pragma`	OpenACC	GPUs and distributed	Low-level parallelism
AHP [26]	`pragma`	CUDA	GPUs and multi-cores	Low-level parallelism
REPARA [5]	C++11 attributes	-	-	High-level parallelism
SPar [9]	**C++11 attributes**	**Internal library**	**GPUs, multi-cores, and distributed**	**High-level parallelism**

3.5 Code generation overhead

In this section we discuss the overhead of the code generation by presenting the performance difference in the total execution time between the code generated by SPar and handwritten code using the same underlying libraries. In the MB benchmark, the code generated by SPar presented lower execution times when compared to the handwritten code: up to 8.5% and 10.8% lower execution times in CUDA for the two workloads, respectively, and up to 18% and 21.2% lower execution times in OpenCL for the two workloads, respectively. In the RT benchmark the code generated by SPar presented slightly higher execution times than the handwritten code: up to 2.4% in the Workload 1 and up to 2.5% in the Workload 2. In the LD benchmark the code generated by SPar presented up to 20% higher execution times when compared to the handwritten code in Workload 1 and up to 18% higher execution times in Workload 2. Further details are discussed in Section 5.7 of [27].

4 RELATED WORK

In this section we present our related work. We selected those studies that provide parallel programming abstractions targeting many-core GPUs. Table 2 summarizes the related work.

Most of available abstractions rely on `pragma`-based compiler directives to express parallelism in sequential code. The OpenMP language specification first provided heterogeneous parallelism capabilities in its version 4.0 and improved their support for offloading in version 4.5 [24]. However, programmers must implement their codes using low-level parallelism such as spawning threads, barrier calls, communication queues, etc. The GPU runtime supported by OpenMP is compiler-dependent, meaning it is limited by GPU architectures that are effectively supported by the target compiler.

OpenACC [23] is another annotation-based tool developed by the industry. The OpenACC language provides some abstractions but it is mostly composed of lower-level concepts such as `worker`, `vector`, and `gang`. The programmer needs to consider hardware configurations and design low-level parallelism strategies in order to efficiently exploit the underlying GPU performance. Moreover, the support for hybrid parallelism is compiler-dependent. For example, the PGI 15.10 compiler for OpenACC supports multi-core parallelism but you cannot combine it with GPU parallelism.

There are also `pragma`-based tools developed by the academia focused in GPU parallelism (hiCUDA) [12] or distributed GPU parallelism (XMP-ACC and XACC) [18, 22]. However, these languages require application programmers to manually deal with memory management and data copies between the host and device memory spaces. Also, some of the aforementioned tools are tied to NVIDIA boards due to their CUDA runtime and none of them provide stream parallelism abstractions. When using low-level parallelism strategies, the parallel code is mixed with the application business logic code. Consequently, debugging and maintaining these applications becomes intricate. Opposite, the SPar programming model targets a clear separation between parallelism and application code.

The Automatic Heterogeneous Pipelining (AHP) framework [26] focuses on identifying the pipeline stages, mapping them into processing units (PUs) of heterogeneous systems, and scheduling their execution. Although the AHP language resembles SPar, we focus on providing efficient and high-level abstractions decoupled from the underlying hardware. Instead, AHP expects the programmer to provide hand-written optimized code for the available PUs.

REPARA was an European project that ran between 2013 and 2016. They promote using C++11 attributes to express stream and data parallelism by annotating parallel patterns in sequential source code [4]. It included annotations for three parallel patterns: Pipeline, Farm, and Map. They also proposed a `target` attribute to indicate heterogeneous computer architectures such as GPU and FPGA, but the support for heterogeneous parallelism was left as future work [4]. Contrary to SPar, the REPARA project never assembled a language interpreter and source-to-source compiler to implement the proposed transformation rules, being a theoretical work.

Among the tools for programming stream processing applications, SPar stands out by its abstraction of the underlying hardware architecture from the programmer's perspective. Our work is the first on providing high-level C++11 annotations as an API that does not require significant code refactoring in sequential programs while enabling multi-core CPU and many-core GPU parallelism exploitation for stream processing applications.

5 CONCLUSION

This paper presented an extension to SPar that allows parallel code generation for CPUs combined with GPUs on stream processing applications. It provides support for exploiting combined stream and data parallelism using C++ attributes in the serial code. Our experiments revealed that employing SPar to generate heterogeneous parallel code targeting CPU and GPU can vastly improve the throughput performance compared to parallel code targeting CPU-only. The heterogeneous GPU code generated by SPar was up to 47% better than the best parallel CPU version in the benchmark LD, 135% in MB (665% when using batching), and 331% in RT. Those results demonstrate that it provides high-level abstractions that hide architecture details from programmers while keeping relevant performance results, even when considering a heterogeneous architecture composed of processors and accelerators like GPUs.

With this research, several opportunities for future work are open. We plan to investigate techniques for supporting multiple GPUs and other accelerators like FPGAs in SPar. Another possibility is exploiting hybrid parallelism with SPar, where the CPU and the

GPU can simultaneously apply data parallelism on stream elements. Also, we could provide runtime optimizations and algorithms to decide if the CPU or the GPU should process a stream element to improve the application's overall performance. This optimization is relevant as not every computation is suitable for GPUs.

ACKNOWLEDGMENTS

We would like to acknowledge the support of LAD-PUCRS, GMAP research group and PUCRS university. This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and FAPERGS 10/2020-ARD project SPar4.0 (N^o 21/2551-0000725-7).

REFERENCES

[1] Marco Aldinucci, Guilherme Peretti Pezzi, Maurizio Drocco, Concetto Spampinato, and Massimo Torquati. 2015. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *The International Journal of High Performance Computing Applications* 29, 4 (Feb. 2015), 461–472. https://doi.org/10.1177/1094342014567907

[2] AMD. 2022. AMD Documentation. Online. https://docs.amd.com/

[3] Christopher Brown, Vladimir Janjic, Adam D. Barwell, J. Daniel Garcia, and Kenneth MacKenzie. 2020. Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++. *International Journal of Parallel Programming* 48, 4 (Aug. 2020), 603–625. https://doi.org/10.1007/s10766-020-00667-x

[4] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. 2018. Data stream processing via code annotations. *The Journal of Supercomputing* 74 (Nov. 2018), 5659–5673. https://doi.org/10.1007/s11227-016-1793-9

[5] M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. 2016. Introducing Parallelism by Using REPARA C++11 Attributes. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'16)*. IEEE, Heraklion, Greece, 354–358. https://doi.org/10.1109/PDP.2016.115

[6] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46, 1 (Feb. 2018), 62–80. https://doi.org/10.1007/s10766-017-0490-5

[7] Milind Girkar and Constantine D. Polychronopoulos. 1992. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems* 3, 2 (March 1992), 166–178. https://doi.org/10.1109/71.127258

[8] Dalvan Griebler. 2016. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism.* Ph.D. Dissertation. Computer Science Department - University of Pisa, Pisa, Italy. https://gmap.pucrs.br/dalvan/papers/2016/thesis_dalvan_UNIPI_2016.pdf

[9] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (March 2017), 1740005. https://doi.org/10.1142/S0129626417400059

[10] Dalvan Griebler and Luiz Gustavo Fernandes. 2017. Towards Distributed Parallel Programming Support for the SPar DSL. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo'17)*. IOS Press, Bologna, Italy, 563–572. https://doi.org/10.3233/978-1-61499-843-3-563

[11] Dalvan Griebler, Adriano Vogel, Daniele De Sensi, Marco Danelutto, and Luiz Gustavo Fernandes. 2019. Simplifying and implementing service level objectives for stream parallelism. *Journal of Supercomputing* 76 (June 2019), 4603–4628. https://doi.org/10.1007/s11227-019-02914-6

[12] Tianyi David Han and Tarek S. Abdelrahman. 2011. *hi*CUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (Jan. 2011), 78–90. https://doi.org/10.1109/TPDS.2010.62

[13] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Otto Grimm. 2014. A catalog of stream processing optimizations. *Comput. Surveys* 46, 4 (March 2014), 46:1–46:34. https://doi.org/10.1145/2528412

[14] Renato B. Hoffmann, Dalvan Griebler, Marco Danelutto, and Luiz G. Fernandes. 2020. Stream Parallelism Annotations for Multi-Core Frameworks. In *XXIV Brazilian Symposium on Programming Languages (SBLP) (SBLP'20)*. ACM, Natal, Brazil, 48–55. https://doi.org/10.1145/3427081.3427088

[15] Renato B. Hoffmann, Júnior Löff, Dalvan Griebler, and Luiz G. Fernandes. 2022. OpenMP as Runtime for Providing High-Level Stream Parallelism on Multi-Cores. *Journal of Supercomputing* 78, 6 (apr 2022), 7655–7676. https://doi.org/10.1007/s11227-021-04182-9

[16] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. 2011. Sponge: Portable Stream Programming on Graphics Engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM, Newport Beach, California, USA, 381–392. https://doi.org/10.1145/1950365.1950409

[17] Ryota Kimura, Masafumi Seigo, Russell A. Chipman, and Seiichiro Kitagawa. 2019. Optical simulation for illumination using GPGPU ray tracing. In *Physics and Simulation of Optoelectronic Devices XXVII*, Bernd Witzigmann, Marek Osiński, and Yasuhiko Arakawa (Eds.), Vol. 10912. International Society for Optics and Photonics, SPIE, San Francisco, 171 – 179. https://doi.org/10.1117/12.2506129

[18] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. 2012. An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters. In *Euro-Par 2011: Parallel Processing Workshops*. Vol. 7155. Springer, Berlin, Heidelberg, 429–439. https://doi.org/10.1007/978-3-642-29737-3_48

[19] Brian Lonsdorf and Matthias Benkort. 2020. Professor Frisby's Mostly Adequate Guide to Functional Programming. GitBook. https://mostly-adequate.gitbooks.io/mostly-adequate-guide/

[20] Júnior Löff, Renato Barreto Hoffmann, Dalvan Griebler, and Luiz G. Fernandes. 2021. High-Level Stream and Data Parallelism in C++ for Multi-Cores. In *XXV Brazilian Symposium on Programming Languages (SBLP) (SBLP'21)*. ACM, Joinville, Brazil.

[21] Michael McCool, Arch D. Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1 ed.). Morgan Kaufmann, 225 Wyman Street, Waltham, MA 02451, USA.

[22] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. 2014. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the 1st Workshop on Accelerator Programming using Directives*. IEEE, New Orleans, 27–36. https://doi.org/10.1109/WACCPD.2014.6

[23] OpenACC-Standard.org 2015. *OpenACC Programming and Best Practices Guide.* OpenACC-Standard.org. https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf

[24] OpenMP Architecture Review Board 2015. *OpenMP Application Programming Interface.* OpenMP Architecture Review Board. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf Version 4.5.

[25] OpenMP Architecture Review Board 2018. *OpenMP Application Programming Interface.* OpenMP Architecture Review Board. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf Version 5.0.

[26] Jacques A. Pienaar, Srimat Chakradhar, and Anand Raghunathan. 2012. Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE, Salt Lake City, United States of America, 1–12. https://doi.org/10.1109/SC.2012.22

[27] Dinei André Rockenbach. 2020. *High-Level Programming Abstractions for Stream Parallelism on GPUs.* Master's Thesis. School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil.

[28] Dinei A. Rockenbach, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo Fernandes. 2019. High-Level Stream Parallelism Abstractions with SPar Targeting GPUs. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo) (ParCo'19, Vol. 36)*. IOS Press, Prague, Czech Republic, 543–552. https://doi.org/10.3233/APC200083

[29] Dinei A. Rockenbach, Charles Michael Stein, Dalvan Griebler, Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2019. Stream Processing on Multi-cores with GPUs: Parallel Programming Models' Challenges. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (IPDPSW'19)*. IEEE, Rio de Janeiro, Brazil, 834–841. https://doi.org/10.1109/IPDPSW.2019.00137

[30] Charles M. Stein, Dinei A. Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz G. Fernandes. 2020. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurrency and Computation: Practice and Experience* na, na (May 2020), e5786. https://doi.org/10.1002/cpe.5786

[31] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. 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*. IEEE, Anchorage, AK, USA, 1176–1182. https://doi.org/10.1109/IPDPS.2011.269

[32] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO '09)*. IEEE, Seattle, WA, USA, 200–209. https://doi.org/10.1109/CGO.2009.20

[33] Dakuan Yu, Wurui Ta, and Youhe Zhou. 2021. Fractal diffusion patterns of periodic points in the Mandelbrot set. *Chaos, Solitons and Fractals* 153 (2021), 111599. https://doi.org/10.1016/j.chaos.2021.111599

[34] GuoQiang Yu and Dong Qiu. 2021. Research on Lane Detection Method of Intelligent Vehicle in Multi-road Condition. In *2021 China Automation Congress (CAC)*. IEEE, Beijing, 2779–2783. https://doi.org/10.1109/CAC53003.2021.9728269