Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200083

High-Level Stream Parallelism Abstractions with SPar Targeting GPUs

Dinei A. ROCKENBACH [a,1], Dalvan GRIEBLER [a,c], Marco DANELUTTO [b] and Luiz G. FERNANDES [a]

[a] *School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre – Brazil.*
[b] *Computer Science Department, University of Pisa (UNIPI), Pisa, Italy.*
[c] *Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio – Brazil*

Abstract. The combined exploitation of stream and data parallelism is demonstrating encouraging performance results in the literature for heterogeneous architectures, which are present on every computer systems today. However, provide parallel software efficiently targeting those architectures requires significant programming effort and expertise. The SPar domain-specific language already represents a solution to this problem providing proven high-level programming abstractions for multi-core architectures. In this paper, we enrich the SPar language adding support for GPUs. New transformation rules are designed for generating parallel code using stream and data parallel patterns. Our experiments revealed that these transformations rules are able to improve performance while the high-level programming abstractions are maintained.

Keywords. Parallel Programming, Domain-Specific Language, C++11 Attributes, Parallel Patterns, Stream Processing, GPGPU, GPU Programming

1. Introduction

Stream processing applications are present in different domains and are receiving renewed attention in the last decade, mostly because of the importance of stream processing in the core of big data and Internet of Things technologies [4]. In addition to that, the ubiquitous presence of parallel hardware architectures [16] led researchers to develop new tools focused on stream parallelism [22,2,9]. In recent studies [23,13,1,21,7], data parallelism has been exploited via proper software extensions to take advantage of the emerging massively parallel architectures such as GPUs (Graphics Processing Units), which were intentionally designed for data parallelism.

Parallel programming libraries [1,21] offer good performance but lower-level programming abstractions. To meet higher-level abstractions, some tools [23,13,7] prefer to focus on compiler techniques to alleviate the parallel programming burden of GPUs. The problem is that they still require code refactoring to properly exploit the parallelism in

[1]Corresponding Author: {dinei.rockenbach, dalvan.griebler}@edu.pucrs.br

stream processing applications. Alternatively to these options, the SPar[2] domain-specific language [9] provides a productive parallel programming model without adding significant performance overheads for multi-cores [11]. Although that SPar is demonstrating a good compromise between productivity and performance among different stream processing applications for multi-core architectures [11,10,12], automatic code generation for heterogeneous architectures composed of CPU and GPU is still not supported. Our recent investigations using SPar to annotate stream parallelism for multi-cores with manually programming data parallelism for GPUs have shown promising performance results [20]. In this paper, we present an extension to SPar language for supporting GPU data parallelism. We also discuss the compiler transformation rules and evaluate them in a set of experiments. Therefore, our contributions may be summarized as follows:

- we introduce new SPar attributes that enrich the expressiveness and semantics of the language;
- we design new compiler transformation rules suitable to implement both stream and data parallel patterns after proper source code annotations;
- we describe experiments aimed at assessing performance of our transformation rules targeting code generation for heterogeneous parallel architectures.

This paper is organized as follows: Section 2 presents the related work. Section 3 presents two new attributes for the SPar language. Section 4 presents the definitions and compiler transformation rules for the *Map* parallel pattern based on the new attributes. In Section 5 we perform a performance evaluation of the transformation rules. Finally, Section 6 present our conclusions and future work.

2. Related Work

Skeleton-based frameworks like FastFlow [2] and DSLs like StreamIt [22] provide different programming approaches and levels of abstraction to developers. Libraries like Intel's TBB (Threading Building Blocks) [18] also offer support to the parallel implementation of stream processing applications by instantiating the *Pipeline* parallel pattern. Support for GPGPU in FastFlow [3,1] focus in *Stencil* parallel pattern but also allows the implementation of *Map*, *Reduce*, and its combinations. Another C/C++ library based on algorithm skeletons/parallel patterns is represented by SkelCL [21]. On SkelCL, user defined kernels are passed as string to the parallel pattern classes like *Map*, *Zip* (a special case of gather [16]), *Reduce*, and *Scan*. These functions are combined with skeleton code to generate the final OpenCL kernels.

StreamIt [22] is a new imperative programming language focused on stream processing applications. There are only the works of [23] and [13] that extended StreamIt to support CUDA code generation and so far is no longer updated. Sarek (Stream ARchitecture using Extensible Kernels) [5] is a customized language for writing GPGPU kernels in the OCaml language. There is also SkePU 2 [7], which provides a source-to-source compiler tool and a parallel runtime. The source code can be compiled by any C++11 compiler to produce a sequential executable. SkePU 2 compiler generates all CUDA and OpenCL kernel code.

[2]SPar home page: https://gmap.pucrs.br/spar

SPar is unique in the stream parallelism domain regarding the level of abstraction and code intrusion. While sharing similar streaming concepts with FastFlow and StreamIt as well as a slightly similar approach as SkePU 2 (that uses C++11 attributes for some advanced features), no other study aims to provide stream parallelism support without requiring code refactoring and restructuring. There are only solutions that aim for sequential code maintainability for data parallelism like OpenMP [6] and OpenACC [17].

3. New Attributes for SPar

SPar offers five standard C++11 attributes that application programmers may use to annotate the source code [14]. Two of them are "identifiers" (ID): `ToStream` annotation delimits the streaming region and `Stage` delimits each of the computation "phases" (or *stages*). The other three are "auxiliary" (AUX) attributes: `Input` and `Output` are used to specify the stream items, while `Replicate` is used to specify the degree of parallelism for a stage. Listing 1 demonstrates how these attributes are used to annotate sequential source code. This example is computing the Mandelbrot Streaming application. `ToStream` marks where the stream parallelism region starts and also refers to the stream generator stage. Inside the stream computation there are two `Stages` annotations identifying the stream operators. The data stream dependencies are specified through the `Input` and `Output` attributes. `Replicate` in line 5 indicates the degree of parallelism for that specific stage, running the amount of replicas given as argument in the attribute. The last `Stage` simply shows line by line the Mandelbrot image. It cannot be replicated because `ShowLine` is a stateful operator.

The current SPar attributes are closely related to the stream parallelism domain. Also, they do not express any semantics of the data parallelism properties. Therefore, we created a novel attribute called `Pure` to be used along with the `Stage` attribute list or as identifier inside `Stage` annotated regions. This attribute indicates that the annotated code block is a pure function, "whose output depends only on its input, and does not modify any other system state" [16]. In SPar, a `Stage` or code block will be considered a pure function when it satisfies the following statements to guarantee correct use and correct code generation:

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

From the programmer perspective, the `Pure` attribute is just another attribute allowing to identify data parallelism inside the `Stage`. On the other hand, the compiler transformation rule identifies that this region/function can be computed in parallel over multiple data. It is up to the compiler decide which parallel architecture (GPU or multi-core) generate the stream parallelism with data parallelism code. Section 4 will describe the design of the compiler transformation rules to target data parallelism for GPUs.

In our previous work, we evaluated different parallel programming models when implementing stream and data parallelism combined [19]. One lesson learned is that fine-grained stream processing may not generate enough workload to properly exploit massively parallel architectures such as GPUs. Thus, some stream processing applications may not provide the expected performance scalability when using GPUs. For these cases, we are providing the possibility to express stream batches in SPar through the new auxiliary attribute for the `ToStream`, named `Batch`. 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 subsequent stages, which can be or not a `Pure` stage. In short, `Batch` will now allow programmers to define the stream item granularity.

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

Listing 1: Mandelbrot Streaming annotated with SPar using the new attributes.

Observe that none of these attributes are actually related to underlying parallel architecture. They were intentionally designed to express data parallelism properties such as data granularity (`Batch`) and single instruction for multiple data (`Pure`). If we compare to existing data parallel programming models such as OpenMP [6], `Batch` has a meaning to OpenMP *chunk* and `Pure` has a meaning similar to OpenMP *parallel for* where every computation inside the region can be performed in parallel and independently. For this work, data parallelism will be purposely exploited in GPUs. However, these new attributes are also open for further investigations and research on multi-core and cluster parallel architectures. The central point is that the programmer is no longer obliged to reason about the parallel architecture details when developing its application such as required by CUDA or OpenCL. SPar's compiler and transformation rules handle this complexities in place of programmers through its high-level annotation-based language.

Listing 1 exemplifies the use of our new attributes in the existing SPar annotations. We are just adding the `Pure` attribute in the `Stage` annotation in line 5 of Listing 1 because the `for` loop in line 6 is a pure function. Moreover, we inserted the `Batch` attribute in line 2, allowing the control of the stream granularity. It is worth point out that the application latency and throughput are directly impacted by the use of this attribute. However, the programmer may test and choose the best configuration (size of the batch) that fits the performance requirements. Section 5 will discuss the performance impacts of these attributes.

4. New Compiler Transformation Rules for SPar

In his PhD Thesis, Griebler [8,9] designed the original structure of the SPar language. The SPar attributes are combined in annotation schemas, which trigger transformation rules in the compiler. These transformation rules are based on previous definitions. We present current SPar definitions and transformation rules for *Pipeline* and *Farm* parallel patterns and built upon those to generate novel definitions and transformation rules for the *Map* parallel pattern.

To express the definitions and transformation rules, Griebler created a particular notation: `ToStream` and `Stage` attributes are represented by T_{id} and S_{id}, where *id* represents a numeric identifier. `Input`, `Output`, and `Replicate` attributes are represented by I_i, O_i, and R_n, respectively. I_i and O_i may contain a list of typed variables a_i, and n denotes the integer number of replicas for `Replicate` argument. To denote a code block with one or more statements it is used $\square_{id}$. The scope of the sentence is denoted by $\{...\}$. The annotations that contain one identifier attribute and optionally a list of auxiliary attributes, are denoted using [[...]] [14].

The current definitions and transformation rules of SPar [9] are generating the stream parallel patterns *Pipeline* and *Farm*. They are implemented in the SPar compiler for transforming the annotated code into C++ code with calls to the FastFlow library, which provide classes and built-in functions for implementing these parallel patterns. Griebler uses functional semantics to define the *Farm* and *Pipeline* patterns: $farm(E,W,C)$ has arguments E (Emitter, the stream item scheduler), W (Worker, that compute stream items), and C (Collector, which gather results/stream items from the workers), where E, C, and W receive a $\square_{id}$ as argument; and $pipe(S_1,S_2,...)$ has two or more stages, which can be $\square_{id}$ or $farm$ instances. The current SPar transformation rules can generate a combination of these patterns based on the annotation schema.

In this paper, we focus in the combination of data stream and data parallel patterns. First, we concentrate only in the *Map* pattern, as it is the simplest and widely used pattern for data parallelism [16]. Using functional semantics, we defined this pattern as: $data = map(\square^{p}_{id})$, where $\square^{p}_{id}$ is the pure function or code wrapper that computes over multiple data independently and transforms them into $data$. This $data$ can be a list, vector, or an array of data.

Before introducing our novel definitions and transformation rules, we extend the previous SPar notation: P_i denotes a `Pure` attribute and $\forall_{id}(\square_{id})$ denotes a `for` statement [14] with a code block. The `Batch` attribute is not discussed in this section since it only changes the data management and does not interferes in the pattern generation.

There are six definitions presented in [9] related to the transformation rules for generating *Pipeline* and *Farm* parallel patterns from SPar annotations. Table 1 presents our new definitions aimed at supporting the transformation rules with the *Map* pattern. The changes with respect to the definitions from [9] are highlighted in blue color.

Table 1. Definitions for transformation rules adapted from [9].

D_0	A *generic stage* ψ is a $\Box$ annotated with S that contains in its attribute list R_n and O_i and therefore requiring a further $\Box$ gathering its results.
D_1	A $\Box$ may appear as a *pipe* stage, as an E or C stage in a *farm* or as the *map* function if its annotation list S does not contain the attribute R_n.
D_2	A $\Box$ where the first statement is a $\forall_{id}$ annotated with a S followed by P in its attribute list becomes a *map*.
D_3	A $\Box$ with an annotation list S containing an R_n attribute may appear as a W stage in a *farm* or as the parameter of a *map*.
D_4	When D_1 and D_2 applies on a $\Box$, a *map* is instantiated as a *pipe* stage.
D_5	When D_2 and D_3 applies on a $\Box$, a *map* is instantiated inside the W stage of the *farm*.
D_6	A $\forall(\Box)$ annotated with only P inside a S's code block becomes a *map* nested into a *pipe*'s S or *farm*'s W.
D_7	T is a *map* when a $\Box$ has $\forall_0$ as the first statement annotated with T, where right after this $\forall_0$ there is only a single $\Box$ which is a $\forall_1$ annotated with S and contains P in its attribute list.
D_8	A T is a *farm* when the first S annotation contains R_n in the attribute list of a maximum two S.
D_9	A T is a *pipe* when the first S does not have R_n in the attribute list or when there are more than two Ss.
D_{10}	A *farm* is a stage of *pipe* when D_7 cannot be applied and $\Box$ is annotated with S that contains R_n in the attribute list.

From the original SPar transformation rules [9], we take the fourth transformation rules as an example to demonstrate the combination of stream and data parallelism. Adding P_i and considering a $\forall(\Box)$ as the code block of the first S in the transformation rule 4 from [9], we can apply D_2 and D_4 to obtain Rule 1. In this case, we combine the *Map* and *Pipeline* patterns. Each stream item produced by the first *pipe* stage instantiate the *map* to exploit data parallelism.

$$[[T_0]]\{\Box_0,[[S_0,P_i]]\{\forall(\Box_1)\}\} \Rightarrow pipe(\Box_0,\ map(\Box_1)) \tag{1}$$

Similarly, if we take transformation rule 3 from [9], add P_i and consider a $\forall(\Box)$ as the code block of the first S, we can apply D_2, D_3, and D_5 to obtain Rule 2. In this case, a new parallel pattern is generated, combining *Farm* with workers instantiating the *Map* pattern.

$$\begin{gathered}[[T_0]]\{\Box_0,[[S_0,O_i,R_n,P_i]]\{\forall_0(\Box_1)\},[[S_1]]\{\Box_2\}\}\\ \Downarrow \\ farm(E(\Box_0),\ W(map(\Box_1)),\ C(\Box_2))\end{gathered} \tag{2}$$

Adding P_i in the fifth rule from [9], with $\forall(\Box)$ as the code block, we can apply D_2, D_3, and D_5 and obtain Rule 3. This Rule combines three parallel patterns: *Pipeline*, *Farm*, and *Map*. The *pipe* is generated based on D_9. The *farm* appears as a *pipe* stage (based on $D_1 0$) and the *map* pattern comprises the *farm*'s worker stage (W), according to D_5.

$$[[T_0]]\{\square_0,[[S_0]]\{\square_1\},[[S_1,R_n,P_i]]\{\forall(\square_2)\}\} \\ \Downarrow \\ pipe(\square_0,\ farm(E(\square_1),\ W(map(\square_2)))) \tag{3}$$

Definition D_6 allows P to be employed as ID attribute, which provides more flexibility to SPar application. If only part of the last `Stage` from Rule 3 is a pure function, P_i could be applied in this specific code block, as demonstrated by Rule 4.

$$[[T_0]]\{\square_0,[[S_0]]\{\square_1\},[[S_1,R_n]]\{\square_2,[[P_i]]\{\forall(\square_3)\}\}\} \\ \Downarrow \\ pipe(\square_0,\ farm(E(\square_1),\ W(\square_2,map(\square_3)))) \tag{4}$$

Rule 5 applies D_7 to generate a single *map* pattern from a T attribute. The presence of the `Pure` attribute in this specific code structure simplifies the implementation and allows the exploitation of a pure data parallelism.

$$[[T_0]]\{\forall_0([[S_0,P_i]]\{\forall_1(\square_0)\})\} \Rightarrow map(\square_0) \tag{5}$$

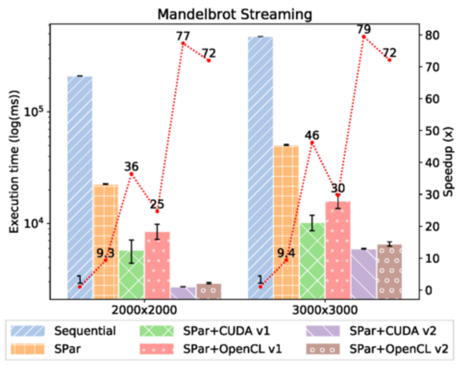
5. Performance Evaluation

To support data parallelism for GPUs in SPar, we decided to generate CUDA and OpenCL code. Although we aim to offer multi-GPU support, we focused in a single GPU in these experiments. The transformation rules were created to generate parallel patterns, however, CUDA and OpenCL does not offer support for a structured parallel programming approach. Therefore, we implemented the *Map* pattern by transforming the pure function $\square_{id}^{p}$ into a GPU kernel, where each thread launched goes throughout this code wrapper. Then, inside this GPU kernel each thread get its global index and computes over a different data index. Consider N as the number of iterations of the annotated $\forall$ and *max_threads* the maximum number of threads per block available in the GPU. On the absence of the `Batch` attribute, we launch N threads divided in $N/max\_threads$ blocks. When the `Batch` attribute is used, we launch $N * batch\_size$ threads on each kernel call. In this case, we modify the previous and next computation stages to generate and consume stream items of size *batch_size*. Prior to implementing the transformation rules in the compiler, we evaluated them by generating the CUDA and OpenCL code manually when our transformation rules were triggered. Therefore, this performance evaluation was carried out by manually performing the work that would be done by the compiler. We want to show that our transformation rules could work in a future compiler implementation.

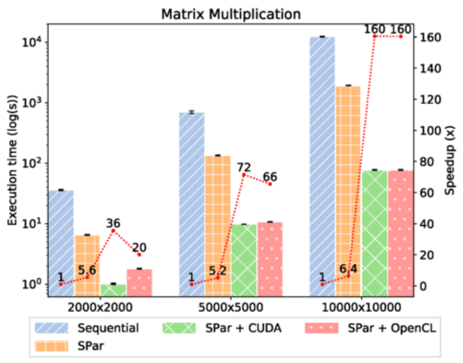
To integrate stream parallelism on the multi-core and data parallelism with CUDA, we added a `cudaStream` object on each stream item to properly define dependencies between data transfer and kernel function calls. For the OpenCL runtime, we added a `cl_kernel`, a `cl_command_queue`, and a `cl_event` object on each stream item. The `cl_kernel` are not thread-safe [15] and must be allocated for each thread. The `cl_command_queue` allows overlapping kernel and memory copies between different stream items and the `cl_event` is used to synchronize asynchronous calls between different pipeline stages.

The experiments ran in a server machine that has an Intel(R) Core(TM) I9-7900X @ 3.3GHz (10 cores and 20 threads), 32GB of RAM memory and two Titan XP GPUs (although we use only one of them in this experiments) with compute capability 6.1 and each one has 12GB of memory. The system was running on Ubuntu OS (kernel 4.15.0-43-generic). All programs were compiled using -O3 compiler flags. The software used were G++ 9.1, NVCC 10.0.130, OpenCL 1.2, SPar, and FastFlow. We chose the best degree of parallelism and batch sizes by empirical testing the applications under different configurations. The SPar implementations ran with 20 worker replicas and versions combining SPar with CUDA or OpenCL ran with 10 worker replicas in the annotated regions with the `Replicate` attribute. Each version was executed five times and the average execution time is plotted, while error-bars show the standard deviation.

We present experiments using two pseudo-applications: Mandelbrot Streaming and Matrix Multiplication. We focused in traditional HPC metrics such as execution time and speedup to observe the applications scalability and performance.



(a) Mandelbrot Streaming.



(b) Matrix Multiplication.

Figure 1. Experiments Results.

We tested two workloads for the Mandelbrot Streaming application: generating 2000x2000 and 3000x3000 fractal images, both with a maximum of 100,000 iterations per single pixel. This fractal image size represent 4,000,000 and 9,000,000 numbers between -2.125-1.5 and 0.875+1.5. The annotation schema presented in [8] ("SPar" in Figure 1a) shows 9.3× and 9.4× of speedup with respect to the sequential version for our workloads. The simplest modification is to insert the `Pure` as auxiliary attribute in the first `Stage` annotation. This annotation schema triggers Transformation Rule 2 and generates the *Farm* with *Map* pattern. This version shows 36× and 46× speedup for the CUDA runtime ("SPar+CUDA v1" in Figure 1a), and 25× and 30× speedup for the OpenCL runtime ("SPar+OpenCL v1") with respect to the sequential times. These versions presented an unexpectedly high standard deviation, which is due to data transfer between CPU and GPU.

As demonstrated by our previous study, a single Mandelbrot line does not generate enough workload to fully utilize the GPU [19]. Therefore, we can add `Batch` attribute in the `ToStream` annotation to achieve further performance improvements, as demonstrated in Listing 1. Using a batch size of 30 for this annotation schema yields 77× and 79× speedup for the CUDA runtime ("SPar+CUDA v2" in Figure 1a), and 72× speedup in both workloads for the OpenCL runtime ("SPar+OpenCL v2"). Each stream item of these batch versions are calculating 30 lines of the Mandelbrot set in a single kernel call. The performance improvement is explained by this batch of lines utilizing the massive parallelism of the GPU.

We discuss here the matrix multiplication presented in [8] as an example of data-parallel algorithms. We ran this experiment with matrices of 2000x2000, 5000x5000, and 10000x10000 32-bit elements. The `Pure` attribute can be added to the `Stage` annotation of [8] to trigger Transformation Rule 5. It generates a single *Map* pattern for this annotation schema.

The SPar annotated version presented in [8] for multi-core architecture ("SPar" in Figure 1b) achieved 5.6×, 5.2×, and 6.4× speedup with respect to the sequential version in our tests. Adding the `Pure` attribute in the `Stage` annotation yields 36×, 72×, and 160× speedup for the CUDA runtime ("SPar+CUDA") in the three workloads. For the OpenCL runtime ("SPar+OpenCL") this modification yielded 20×, 66×, and 160× speedup.

6. Conclusion

In this paper, we enriched the expressiveness of the SPar language to target data parallelism for GPUs, which can in the future be extended to multi-core and cluster architectures. After, we created new compiler transformation rules for generating the *Map* parallel pattern along with the existing stream parallel patterns. Lastly, we carried out a performance evaluation using two pseudo-applications. The outcome is that the language simplicity was maintained (Listing 1) while performance improvements were obtained with respect to only generating parallel code to multi-core without data parallelism support (Figure 1). Using this work's transformation rules, we obtained very similar performance results with respect to our previous work [20], where these applications were fine tuned and manually programmed.

We aim in the future add new definitions and transformation rules that can potentially support more data parallel patterns. We also intend to implement these transformation rules in the SPar compiler to automatically generate GPU parallel code based on our high-level annotations. Given the many challenges of GPU programming, we intend to propose or use an intermediate library such as SkePU and SkelCL to support functional data parallel patterns for GPUs in C++ that are fully compatible with stream parallelism to alleviate the compiler work.

Acknowledgment This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications", the FAPERGS 01/2017-ARD project called PARAELASTIC (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq N 28/2018 project called SParCloud (No. 437693/2018-0). We also thank LARCC for the computing resources.

References

[1] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *FastFlow: high-level and efficient streaming on multi-core*, chapter 13, pages 261–280. John Wiley & Sons, 1st edition, 2017.

[3] M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Applications (IJHPCA)*, 29(4):461–472, 2015.

[4] H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, New York, USA, 2014.

[5] M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4):583–600, Aug. 2014.

[6] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan. 1998.

[7] A. Ernstsson, L. Li, and C. Kessler. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80, Feb. 2018.

[8] D. Griebler. *Domain-Specific Language & Support Tools for High-Level Stream Parallelism*. PhD thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, June 2016.

[9] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, Mar. 2017.

[10] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707. IOS Press, Sept. 2017.

[11] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, pages 1–19, Feb. 2018.

[12] D. Griebler, R. B. Hoffmann, J. Loff, M. Danelutto, and L. G. Fernandes. High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications. In *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 16–27. SBC, October 2017.

[13] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. 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, pages 381–392. ACM, 2011.

[14] ISO/IEC. *ISO/IEC 14882:2017 - Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, 5 edition, Dec. 2017. https://www.iso.org/standard/68564.html.

[15] The Khronos Group. *The OpenCL Specification*, Oct. 2018. v2.2-8.

[16] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[17] OpenACC-Standard.org. *The OpenACC® Application Programming Interface*, Nov. 2018. v2.7.

[18] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Series. O'Reilly Media, 2007.

[19] D. A. Rockenbach, C. M. Stein, D. Griebler, G. Mencagli, M. Torquati, M. Danelutto, and L. G. Fernandes. Stream Processing on Multi-Cores with GPUs: Parallel Programming Models' Challenges. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019.

[20] C. M. Stein, D. Griebler, M. Danelutto, and L. G. Fernandes. Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, Italy, February 2019. IEEE.

[21] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182. IEEE, May 2011.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In R. N. Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer.

[23] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, Mar. 2009.