2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

# Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs

Charles Michael Stein*, Dalvan Griebler*†, Marco Danelutto‡, and Luiz Gustavo Fernandes†
*Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil.
†School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
‡Computer Science Departament, University of Pisa (UNIPI), Pisa, Italy.
***Corresponding author**: dalvan.griebler@acad.pucrs.br

***Abstract*—GPUs have been used to accelerate different data parallel applications. The challenge consists in using GPUs to accelerate stream processing applications. Our goal is to investigate and evaluate whether stream parallel applications may benefit from parallel execution on both CPU and GPU cores. In this paper, we introduce new parallel algorithms for the Lempel-Ziv-Storer-Szymanski (LZSS) data compression application. We implemented the algorithms targeting both CPUs and GPUs. GPUs have been used with CUDA and OpenCL to exploit inner algorithm data parallelism. Outer stream parallelism has been exploited using CPU cores through SPar. The parallel implementation of LZSS achieved 135 fold speedup using a multi-core CPU and two GPUs. We also observed speedups in applications where we were not expecting to get it using the same combine data-stream parallel exploitation techniques.**

## I. Introduction

A significant amount of stream processing applications have been developed in the last decade [1], [2]. The Internet of Things (IoT) is one of the main reasons of the raising importance of stream processing since the number of devices and users exchanging information and communicating to each other have grown exponentially. In addition, also the amount of data generated and transferred through the network increased as well as the need to store and process information in real-time. Data compression applications are of paramount importance for this new IoT and stream processing scenario in order to save storage space. Usually, data compression is done online in backup and storage systems [3] continuously receiving data from one or many data sources. The computational cost for several state-of-the-art compression algorithms is high [4], [5], [6], [7], [8], and the use of high-performance hardware allows Quality of Service (QoS) to be achieved in these applications.

High-performance computing systems are no longer restricted to highly specialized super computing centers. For instance, a single box such as an Nvidia's DGX-2[1] offers high computational power and consumes less energy than several traditional computer clusters. These kinds of computer architectures combine multi-core (CPU - Central Processing Unit) and many-core (GPU - Graphics Processing Units). The efficient targeting of this heterogeneous and highly parallel hardware still represents a challenging task, taking into account that it requires different programming frameworks.

[1]https://www.nvidia.com/en-us/data-center/dgx-2/

General-purpose libraries and domain-specific language (DSL) for parallel programming haven been designed and implemented to help programmers in this non trivial task [9]. Stream parallel programming frameworks targeting multi-cores include languages such as StreamIt [10], libraries that offer stream parallel patterns such as FastFlow [11], GrPPI [12], and TBB [13], and DSL that offer abstractions trough annotations like SPar [14]. To the best of our knowledge, there is no specific stream parallelism abstraction targeting GPUs. The most suitable options to target GPUs are low-level libraries such as CUDA [15] and OpenCL [16], and libraries that offer data parallel patterns such as SkePU [17] and SkelCL [18]. In this paper, we choose SPar because it offers higher-level abstractions for stream parallelism in multi-cores and it has never been used with GPU or with CUDA and OpenCL.

To represent the data compression application domain, we chose Lempel-Ziv-Storer-Szymanski (LZSS) [19]. Our motivation is to investigate how stream parallelism can be implemented in data compression application for parallel computer architectures with multi-core CPU and GPU. In the literature, the parallelism exploitation for multi-core CPU and GPU in LZSS have been separately studied without considering the stream processing scenario and the possibility to use multi-GPUs [5], [6], [7], [8], [3]. The main goal of this work is to show how to improve performance and scalability for data compression applications and to discuss the associated limitations. In particular, the main contribution consists in:

- New parallel algorithms supporting the implementation of LZSS applications in parallel on multi-core CPUs *and* multi-GPUs.
- Experiments demonstrating the feasibility and scalability of the new parallel algorithms with different parallel programming frameworks (SPar, CUDA, and OpenCL).

This paper is organized as follows. Section II discusses relevant related work. Section III describes the new parallel algorithms for implementing stream parallelism in LZSS with multi-core and multi-GPU support. Subsequently, Section IV highlights the performance achieved and discusses the limitations. Finally, Section V concludes this paper and outlines future work perspectives.

2377-5750/19/$31.00 ©2019 IEEE
DOI 10.1109/PDP.2019.00044

## II. Related Work

LZSSPrevious studies implemented LZSS in parallel exploiting GPUs.Our research focused on the stream parallelism exploitation on CPU (multi-core) combined with data parallelism exploitation on single or multiple GPUs.

Ozsoy et. al. have exhaustively studied different ways to accelerate the LZSS application on GPUs, achieving a speedup of 34 [7], [20], [8]. The goals were to exclusively use CUDA and Nvidia hardware as well as to implement the parallelism in GPU for the substring matching and encoding operations. Only pre-processing and post-processing were implemented in parallel on the CPU using POSIX threads. The parallel algorithms designed did not consider stream parallelism and multi-GPUs environments as we did in our studies. A sort of streaming data compression was implemented in [20], were a pipeline was structured inside the GPU code without significant performance improvements. In their implementations, the original substring matching algorithm required additional modifications to allow the parallelism support. Ozsoy et. al. proposed two versions for it: 1P) the input data are read chunk by chunk and assigned to a block of threads to be compressed in parallel; and 2) each GPU thread searches the longest match in the chunk and compression is performed in the CPU. In addition to our stream parallelism implementation, we followed [7], [8]'s second approach in the substring matching algorithm. Our algorithm is different since we do not optimize the control-flow divergence. We used a different strategy for searching the longest match, and we provided the multi-GPU parallelism support. Moreover, we achieved better performance and compression ratios (same compression efficiency as the original serial version).

Zhou et. al. developed Parallel Matching Lempel-Ziv-Storer-Szymanski (PMLZSS) using CUDA [6]. The main goal was to avoid branch divergence and increase performance. The paper proposed a new strategy called matrix matching for this compression application. However, their algorithm lead to a poor compression ratio compared to the original version as well as to poor performance compared to [7], [8]'s and to our work. In addition, neither stream parallelism nor usage of multiple GPU were considered. Another issue reported was that their implementation uses an extra amount of memory.

[5] implemented a better version of [7]. Zu et. al. identified the drawbacks of the previous algorithm and created the GLZSS. They implemented a strategy using hash-table to reduce the algorithmic complexity and accelerate the locating of duplicated substrings. The drawback of their algorithm is the length of the matches, which are limited by the length of the stored data in the hash-table. This reduces the compression ratio of the algorithm. According to [5], they achieved up 2x better performance than [7]. However, the actual speedup was about 4.6 in their machine and [7]'s code was 2.2 faster than the CPU serial version. Note that Zu et. al. do not support stream parallelism, nor they support multi-GPUs, while achieving lower performance and efficiency with respect to our work (see Table I).

In Table I, we compare the related studies regarding the compression ratio in GPU (smaller is better), the best GPU speedup, the parallel programming framework used, and the machine processing settings. This table highlights the achievements and contributions of our work. We improved the compression efficiency in the parallel version, improved the performance (speedup of 72.3 for single GPU and 135.9 for multi-GPU), our code has multi-GPU scalability support, platform portability support (not only Nvidia GPUs) with different parallel programming frameworks (OpenCL, CUDA, and SPar). Our experiments ran in a more sophisticated machine environment, but unfortunately, we were not able to run the codes from other related works to achieve more precise comparisons as these codes were not available and none of the authors replied to our contacts.

TABLE I: State-of-the-art performance comparison for LZSS.

| Work | Ratio | GPU | Tool | Machine |
|---|---|---|---|---|
| [7] | 62% | 34x | CUDA | GPU GTX 480,CPU Intel i7 CPU 920 2.67GHz |
| [6] | 87% | 16x | CUDA | GPU C2070, GTX480, GTX580. Intel i7 990x |
| [5] | 38% | 4.6x | CUDA | CPU AMD A8-3870, GPU GTX590 |
| Our work | 31% | 135.9x | SPar,CUDA, OpenCL | 2 GPUs Titan Xp, Intel Core I9-7900X 3.30GHz |

## III. Stream Parallelism Implementation

LZSS [19] is a compression algorithm from the Lempel-Ziv family and has been used in many compression applications like RAR and PkZip [21]. This compression application uses previous data as a dictionary to find similar data occurrences. Every character is searched in the data already read from the file. When an occurrence is found in the dictionary, called sliding window, we just save the reference and length of this piece of data in the sliding window. In addition to the four characters in the compressed file, there are encoded data. Once the sliding window is filled during the input data reading, we can reproduce the original file by reading the result file and remounting the sliding window so that it can be decompressed.

The sliding window has a fixed size ($WSIZE$ in Algorithm 1) in the LZSS application, which is 4096 bytes. Each interaction of the algorithm updates the sliding window adding the last processed byte and removing the 4096th byte. The size of the search has until 4 bytes to find a match. To this search work, the next 4 bytes must be loaded too, which is the uncoded lookahead that is updated for every new byte read. Therefore, the sliding window is used by the LZSS algorithm to find the longest match of the uncoded lookahead. In Figure 1, we illustrate how the slide window is working with a reduced size. The window size has 4 characters and the uncoded lookahead contains 2 characters. Each epoch presented is one interaction of the LZSS algorithm.

The sequence of operation in the LZZS application can be structured as a pipeline with three potential stages, which are described as follows:

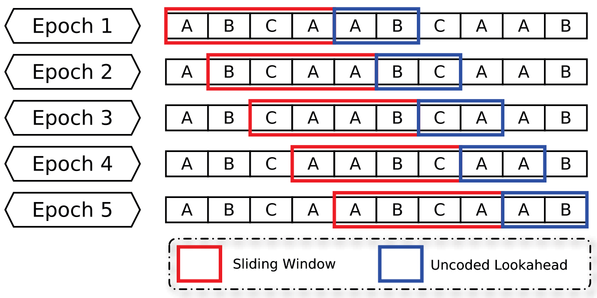

Fig. 1: LZSS sliding window behavior.

- **Read**: reads byte per byte from the input file into data chunks with the size of $BSIZE$ (see Algorithm 1) for sending to the next stage.
- **Find Match**: receives one data chunk per time and searches the longest match (limited by $MAX$ in Algorithm 1), saving the sliding window reference and length of the match to send to the next stage. It also keeps the sliding window and the uncoded lookahead updated.
- **Write**: receives the result of the match to test how to write the result in the file. If the match length was greater than 1, the encoded data is written in the file. Otherwise, only the unchanged bytes are written in the file.

To introduce stream parallelism with SPar[2] for CPUs, we only annotated the code as illustrated in Figure 2. We used SPar because it was specifically designed to simplify the stream parallelism exploitation in C++ programs for multi-core systems [14]. It offers a standard C++11 annotation language to avoid sequential source code rewriting and the SPar compiler generates parallel code using source-to-source transformation techniques. The `ToStream` attribute represents the beginning of a stream region, which tags where a stream parallel region starts in a given program. The `Stages` are defined inside the `ToStream` region to tag the computing phases where stream items will be processed, like workstations in an assembly line. `Input` or `Output` attributes are inserted to define the input and output data dependencies. The attribute arguments can be one or more variables from different data types, which are the stream items that will be consumed or produced by a given region. Finally, the `Replicate` attribute may be inserted in the attribute list of `Stage` to define the degree of parallelism in this region.

Observe that we replicated the **Find Match** stage such that each replica can find the matches for each data chunk. However, the original serial **Find Match** cannot execute in parallel due to the updates in the sliding window for each new byte. To overcome this limitation, we modified the **Find Match**. We can see in Figure 1 that the sliding window is always made of the N bytes read before the current byte being searched, where N is the sliding window size. We keep the sizes of the original algorithm and load a larger data chunk in order to enable the search for every byte in parallel. After, we can process the whole data chunk in the last stage. However, this modification let the algorithm heavier since it will perform unnecessary searches. For example, in Figure 1, the algorithm would not require to run the 2nd and 4th epoch, because the matches at the 1st and 3th epoch would already have been encoded the next characters.

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

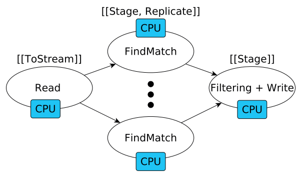

Fig. 2: *LZSS* with SPar.

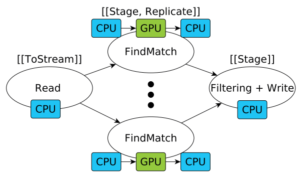

Fig. 3: SPar with GPU.

Algorithm 1 is our implementation of the *FindMatch* procedure to support parallel execution through the elimination of the sliding window. The *lhead*(uncoded lookahead) is created from the data chunk (*batch*) itself in line 5 and 6. Instead of the search performed in the sliding window array, it operates over the item that would have been in the sliding window in line 7 and 9. In order to avoid data losses, we have to transform the offset of the match (result) in line 13 such as it would have been transformed performing in the sliding window original algorithm. In line 14 and 15, we save the *offset* and *length* of each longest match inside a list to send to the next stage. This list is relative to each data chunk to be compressed. We also implemented in the next stage a filtering operation that ignores duplicated encoded data.

**Algorithm 1** Finding Match modified

```
1:  $WSIZE \leftarrow 4KB$, $BSIZE \leftarrow 1MB$, $MAX \leftarrow 15$
2:  procedure FINDMATCH($batch, bindex$)
3:    for each integer $i$ until $BSIZE$ do
4:      $index \leftarrow WSIZE + i$
5:      for each integer $j$ until $MAX$ do
6:        $lhead[j] \leftarrow batch[index + j]$
7:      for each integer $b$ until $WSIZE$ do
8:        $k \leftarrow 0$
9:        while $k \leq MAX \wedge lhead[k] \equiv batch[i + b + k]$ do
10:         $k{+}{+}$
11:       if $k > length$ then
12:         $length \leftarrow k$
13:         $offset \leftarrow (bindex + i)\%WSIZE + j$
14:     $matchOffset[i] \leftarrow offset$
15:     $matchLength[i] \leftarrow length$
```

Once stream parallelism was implemented by using SPar, we implemented a parallel algorithm to exploit data parallelism inside the *FindMatch* procedure for GPU. Algorithm 2 presents our strategy for using both CUDA and OpenCL libraries along with SPar. This algorithm is the kernel code being offloaded to the GPU. Each thread executes a search for one item in the *batch*. We guarantee the thread mapping by getting the global index of the threads in line 2.

**Algorithm 2** Finding Match for GPU

```
1:  $WSIZE \leftarrow 4KB,\ BSIZE \leftarrow 1MB,\ MAX \leftarrow 15$
2:  procedure KERNEL($batch, bindex, mOffset, mLength$)
3:    $i \leftarrow getGpuGlobalIndex()$
4:    $index \leftarrow WSIZE + i$
5:    for each integer $j$ until $MAX$ do
6:      $lhead[j] \leftarrow batch[index + j]$
7:    for each integer $b$ until $WSIZE$ do
8:      $k \leftarrow 0$
9:      while $k \leq MAX \wedge lhead[k] \equiv batch[i + b + k]$ do
10:       $k{+}{+}$
11:     if $k > length$ then
12:       $length \leftarrow k$
13:       $offset \leftarrow (bindex + i)\%WSIZE + j$
14:   $mOffset[i] \leftarrow offset$
15:   $mLength[i] \leftarrow length$
```

Figure 2 illustrates the parallel activity graph when implementing stream parallelism with SPar on GPUs. The second stage has been replicated as many times as the number of GPUs available in the target machine. SPar itself automatically creates on the CPU one thread per replica to manage each used GPU. Consequently, the application may take advantage of both single or multi-GPU to increase the scalability. Data management is not represented in our algorithms, however, we implemented it inside the replicated stage. Since SPar automatically generates a round-robin scheduler, no extra scheduling implementation was required. Moreover, SPar automatically implements data reordering when replicating the stages, thus no extra implementation was required due to the multi-GPU parallelism support.

Our parallel algorithms are generic enough to be used along with other stream-based parallel programming frameworks such as FastFlow, TBB, and GrPPI. Also, the stream parallelism strategies developed in this work for combining CPU and single/multi-GPU could be extend to other data compression applications [22] or real-world stream processing applications [23].

## IV. Experiments

Our performance evaluation was performed in a computer machine equipped with two GPUs Titan XP (each one with 12GB of RAM), an Intel(R) Core(TM) i9-7900X CPU (20 threads with hyper-threading), 16GB of RAM, 2TB of storage, Ubuntu Server 18.04 operating system (Kernel 4.15.0-38-generic). The source codes for all implementations are available online[3]. For the version using only SPar, we ran samples using 1 to 20 replicas. The applications' source code were compiled using `-O3` flag, g++ 7.3.0, nvcc V10.0.130, CUDA 9.2, OpenCL 1.2, and SPar[4]. In the versions with SPar and CUDA/OpenCL, we ran samples from 1 to 2 replicas, which represent the number of GPUs available in the target machine. Each sample was repeated 10 times to compute the speedup and standard deviation. The speedup is computed over the sequential version of the application source code running in the CPU. In our experiments, we also implemented two ways to load and store our workloads. The default mode is to read from and write to the disk. The second mode is to read from and write to the RAM memory. We used the following real-world data sets: **Linux Source**[5] that is a tar file of 797.57MB from the Linux kernel version 4.16-rc4, and **Silesia**[6] is a corpus of data (total of 202.13MB) that represents real-world files (XML, DLLs, and many others).

We present the efficiency of the data compression for the parallel versions on CPU and GPU in Table II. The compression ration is the same after adding parallelism support and modifying parts of the code. In the related studies (Section II), the parallel code version was always less efficient concerning the compression ratio. During the design of our parallelism strategies, we were seeking for efficiency.

TABLE II: LZSS data sets overview (the smaller is the best).

| **Dataset** | **File Size** | **Ratio with CPU** | **Ratio with GPU** |
|---|---|---|---|
| Silesia | 202.13MB | 44.56% | 44.56% |
| Linux Source | 797.57MB | 31.26% | 31.26% |

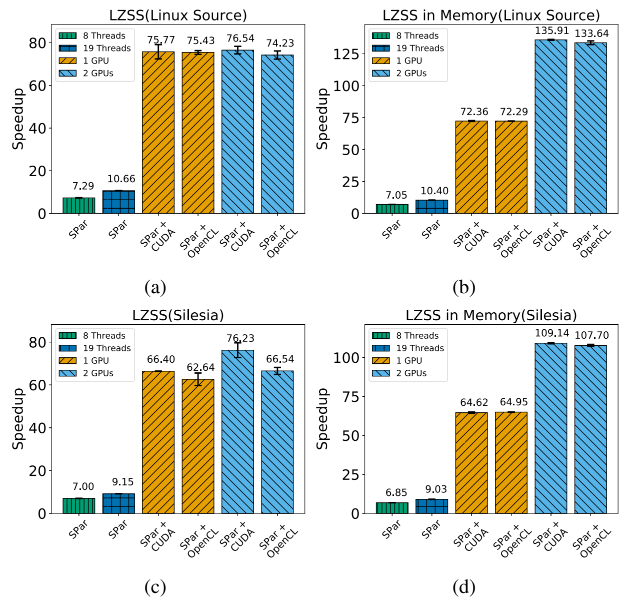

Fig. 4: Performance for the LZSS application.

Figure 4 presents the graphs of the performance results achieved in different circumstances and parallel implementation versions for the LZSS application. Figure 4(a) and 4(b) are showing the results using the **Linux Source** workload while Figure 4(c) and 4(d) are showing the results using the **Silesia** workload. We observe that when using SPar only,

[3]Source codes: https://github.com/larcc-group/lzss-gpu-stream-parallelism
[4]Downloaded from https://github.com/dalvangriebler/SPar
[5]Available in https://www.kernel.org/
[6]Available in http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia

our best case scalability was limited to a speedup of 10.66 (Figure 4(a)). This is a good result if we consider that the machine have actually 10 physical cores available. Yet in Figure 4(a), we achieved good results working with a single replica for the **Find Match** stage, using a single GPU. Since that the data is loaded and stored in the disk for this scenario, we can observe that the performance is not scaling as expected using two GPUs. This is due to the disk bottleneck. The data loading is not fast enough to achieve the GPU processing capacity. In Figure 4(b), when loading and storing data in memory, the performance scales as expected for 2 GPUs. However, our speedup for single GPU has dropped due to the sequential version that is also faster when loading and storing in memory (Figure 4(b) and 4(d)).

The performance results were very similar among CUDA and OpenCL frameworks in the **Linux Source** workload. The only big difference among these frameworks was for the **Silesia** workload in Figure 4(c), which was loading and storing data in the disk. However, there are higher standard deviations in the samples. **Silesia** is a smaller workload than **Linux Source**, which explains the lower speedups for **Silesia** workload. Again, Figure 4(d) highlights the disk bottleneck in Figure 4(c) when using 2 GPUs. Finally, these experiments revealed the great importance of the combined exploitation of stream parallelism and GPUs to improve the performance in the LZSS application (increase 135.91 times in the best case scenario).

Although we achieved satisfactory results with these initial version, there are opportunities for improving the performance results in this application. As an example, we plan to modify the current code to try to eliminate branch divergence in the GPU kernel algorithm, which may come at the price of a decreased compression efficiency. Also, memory management in GPU could be improved with data coalescing.

## V. Conclusion

This paper discussed the combined exploitation of multi-core CPUs and GPUs in the implementation of LZSS applications, using different parallel programming frameworks (SPar, CUDA, and OpenCL).The performance achieved is much better than the one achieved with state-of-the-art implementations. Additionally, our experiments revealed that multi-GPU scalability in LZSS may be not so good when loading and storing data in the disk. In the future, we plan to run experiments in a machine such as Nvdia's DGX-2, which runs with 16 GPUs and to find opportunities to optimize the parallelism exploitation.

## Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, by the EU H2020-ICT-2014-1 project RePhrase (No. 644235), and by the FAPERGS 01/2017-ARD project ParaElastic (No. 17/2551-0000871-5). We thank Nvidia's GPU Grant program for the GPUs donation.

## References

[1] W. Thies and S. Amarasinghe, "An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design," in *International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. Austria: ACM, Sep 2010, pp. 365–376.

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

[3] K. Suttisirikul and P. Uthayopas, "Accelerating the Cloud Backup Using GPU Based Data Deduplication," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2012, pp. 766–769.

[4] S. Kodituwakku and U. Amarasinghe, "Comparison of Lossless Data Compression Algorithms for Text Data," *Indian journal of computer science and engineering*, vol. 1, no. 4, pp. 416–425, 2010.

[5] Y. Zu and B. Hua, "GLZSS: LZSS Lossless Data Compression Can Be Faster," in *Proceedings of Workshop on General Purpose Processing Using GPUs*. Salt Lake City, UT, USA: ACM, 2014, pp. 46:46–46:53.

[6] B. Zhou, H. Jin, and R. Zheng, "A High Speed Lossless Compression Algorithm Based on CPU and GPU Hybrid Platform," in *International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, Sep 2014.

[7] A. Ozsoy, M. Swany, and A. Chauhan, "Optimizing LZSS Compression on GPGPUs," *Future Generation Computer Systems*, vol. 30, pp. 170 – 178, 2014.

[8] A. Ozsoy and M. Swany, "CULZSS: LZSS Lossless Data Compression on CUDA," in *International Conference on Cluster Computing*. IEEE, Sep 2011.

[9] M. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. MA, USA: Morgan Kaufmann, 2012.

[10] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *11th International Conference on Compiler Construction*, ser. CC '02. Grenoble, France: Springer-Verlag, April 2002, pp. 179–196.

[11] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High Level and Efficient Streaming on Multicore*. Wiley-Blackwell, 2014, ch. 13, pp. 261–280.

[12] D. del Rio Astorga, M. F. Dolz, J. Fernndez, and J. D. Garca, "A Generic Parallel Pattern Interface for Stream and Data Processing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, May 2017.

[13] J. Reinders, *Intel Threading Building Blocks*. Sebastopol, CA, USA: O'Reilly, 2007.

[14] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 1740005, March 2017.

[15] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2013.

[16] D. Schaa, D. P. Zhang, P. Mistry, and D. R. Kaeli, *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[17] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 62–80, feb 2018.

[18] 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*. IEEE, may 2011, pp. 1176–1182.

[19] Michael Dipperstein, "LZSS (LZ77) Discussion and Implementation," 2018, last access in Oct, 2018. [Online]. Available: http://michael.dipperstein.com/lzss/index.html

[20] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 37–44.

[21] D. B. David Salomon, G. Motta, *Data compression: The Complete Reference*, 3rd ed. Springer, 2007.

[22] 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*. Campinas, SP, Brasil: SBC, October 2017, pp. 16–27.

[23] 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*, pp. 1–19, Feb 2018.