Received: 18 September 2019 | Revised: 21 January 2020 | Accepted: 3 March 2020

DOI: 10.1002/cpe.5786

WILEY

SPECIAL ISSUE PAPER

Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units

Charles M. Stein[1] | Dinei A. Rockenbach[1,2] | Dalvan Griebler[1,2] | Massimo Torquati[3] | Gabriele Mencagli[3] | Marco Danelutto[3] | Luiz G. Fernandes[2]

[1]Laboratory of Advanced Research on Cloud Computing, Três de Maio Faculty (SETREM), Três de Maio, Brazil

[2]School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil

[3]Computer Science Department, University of Pisa, Pisa, Italy

Correspondence
Dalvan Griebler, School of Technology, Pontifical Catholic University of Rio Grande do Sul, Ipiranga Avenue, Porto Alegre, Rio Grande do Sul 6681, Brazil.
Email: dalvan.griebler@edu.pucrs.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 437693/2018-0; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: 001; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, Grant/Award Number: 17/2551-0000871-5

Summary

Stream processing is a parallel paradigm used in many application domains. With the advance of graphics processing units (GPUs), their usage in stream processing applications has increased as well. The efficient utilization of GPU accelerators in streaming scenarios requires to batch input elements in microbatches, whose computation is offloaded on the GPU leveraging data parallelism within the same batch of data. Since data elements are continuously received based on the input speed, the bigger the microbatch size the higher the latency to completely buffer it and to start the processing on the device. Unfortunately, stream processing applications often have strict latency requirements that need to find the best size of the microbatches and to adapt it dynamically based on the workload conditions as well as according to the characteristics of the underlying device and network. In this work, we aim at implementing latency-aware adaptive microbatching techniques and algorithms for streaming compression applications targeting GPUs. The evaluation is conducted using the Lempel-Ziv-Storer-Szymanski compression application considering different input workloads. As a general result of our work, we noticed that algorithms with elastic adaptation factors respond better for stable workloads, while algorithms with narrower targets respond better for highly unbalanced workloads.

KEYWORDS

data compression algorithms, dynamic reconfiguration, parallel programming, service level objective, stream parallelism, stream processing

1 | INTRODUCTION

Streamed data compression algorithms are used in several real-world systems such as data storage services, web protocols, and in many mobile ad hoc network protocols. In the context of Internet of things (IoT), data compression is frequently used to improve data transferring[1] due to limited network bandwidth. Based on the specific technological scenario where data compression is employed, it might require different nonfunctional requirements concerning either performance (latency and/or throughput) or power consumption[2] (eg, when data compression is applied on embedded devices). Such nonfunctional requirements can often be satisfied by leveraging heterogeneous multicores equipped with accelerators specialized in compression activities.[3] When such specialized coprocessors are not available, graphics processing units (GPUs) still represent interesting candidates for offloading data compression tasks, also because they are becoming popular in IoT embedded systems.[4] However, the efficient exploitation of GPU devices is considered challenging in the context of stream processing,[5] in particular when the target performance metric to

Abbreviations: FAF, fixed adaptation factor; GPU, graphics processing unit; LZSS, Lempel-Ziv-Storer-Szymanski; MBAF, multiplier-based adaptation factor; PBAF, percentage-based adaptation factor; SLO, service level objective.

optimize is latency. This is because GPUs generally work well when a high volume of data is available at the same time, condition that in stream processing applications can be obtained by properly batching input data before processing them as a whole.

Several studies[6-8] proposed new parallel data compression algorithms with the aim of maximizing GPU usage. One of the most effective ways to increase the system throughput is to organize the computation as a pipeline where one or more stages offload batches of input data elements into the GPU. This approach has been pursued in our previous work for the Lempel-Ziv-Storer-Szymanski (LZSS) compression algorithm.[8] We observed that the size of the offloaded batch plays an important role having a significant impact both on the system throughput and on its end-to-end latency.[5] The ideal batch size depends on several factors, among them, the current workload and the input data rate. The effect of using large batches is generally to increase the system throughput at the expense of higher latency due to a longer buffering phase.[9]

In real stream processing scenarios, the optimal microbatch size, that is, the one achieving the desired trade-off between throughput and latency, is not a constant value. Workload fluctuations on stream processing applications are very common, changing the frequency at which new inputs are received by the system and the average computational cost of the processing of each individual input element. This problem can be tackled using dynamic techniques that adapt the microbatch size based on the measured behavior of the system with the goal of achieving a desired service level objective (SLO).[2] Generally, we intend SLO in this article has an acceptable value for a given configuration parameter that keep the average latency within a desired region, that is, lower/higher than a maximum and a minimum threshold identified by the user.

It is widely acknowledged that implementing dynamic adaptations to meet specific SLOs is a complex problem to face. GPU accelerators add further complexity with respect to CPUs-only dynamic adaptation,[10-12] because the number of underlying parameters to consider become greater. In GPU-based systems, the dynamic thread creation and synchronization can add significant overheads because the hardware has been mainly designed for data parallelism, where a batch of data is processed synchronously by many lightweight threads. Furthermore, most of the current heterogeneous systems are based on PCI interconnects that make the overhead of data transfers significant if not properly dealt with References 13,14. Only a small number of research articles implemented adaptive microbatching for stream processing applications (a discussion of them is presented in Section 2). A significant fraction of the proposed algorithms and strategies were designed to reach the maximum performance as possible on a target platform primarily in terms of sustained input rate (throughput). For example, GASSER[15] automatically tunes the microbatch size to optimize throughput. However, it does this by assuming a stationary workload (eg, stable input rate) and once the optimal size has been found it is never recomputed.

In this work, we study the problem of providing adaptive microbatching solutions. To this end, we focus on data compression applications (and in particular to LZSS) due to their importance in IoT domains. Our idea is that the application user is in charge of expressing the desired SLO latency requirements, and the underlying system is able to meet such requirements automatically over the entire computation. To do that, we use a mix of reactive adaptation approaches and closed-loop control algorithms that represent the core contribution of this article. Finally, we validate the proposed techniques using real-world workloads in order to provide a credible analysis.

This article is structured as follows: Section 2 discusses the related work. Section 3 describes the scenario and the design choices, the streamed data compression application adopted, our strategy based on a dynamic loop control for adapting microbatches at run-time, and the four reactive algorithms proposed. Section 4 provides a description of the experiments conducted by using different workloads and the evaluation of the obtained results Finally, Section 5 concludes this article and briefly presents future directions.

2 | RELATED WORK

When reviewing the literature, we identified only few works tackling the problem of microbatch adaptation in stream processing applications. Some of them focused on multicore architectures,[9,11,12,16] while others tackled a similar problem for window-based streaming operators on heterogeneous systems equipped with GPUs.[15,17] To the best of our knowledge, this article presents the first attempt to provide algorithms for the dynamic adaptation of the microbatch size for stream processing applications running on GPU-based systems to meet a user-defined target latency requirement.

Das et al[9] explored the performance impacts of the microbatch size. They aimed at increasing the robustness of the application by adapting the batch to the changes at run-time. Their strategy was based on the fixed-point iteration method to find the intersection between batch processing time and batch interval. However, the algorithm assumes the existence of a specific batch interval where the processing rate matches the data input rate. If there is no such point, the algorithm will not converge with important implications on the application's performance (unbounded running time to find a stable solution).

Dynamic block and batch sizing algorithm (DyBBS)[16] is a proposed technique that tries to minimize the end-to-end latency of batched stream systems, by dynamically adjusting the block and batch size based on the input rate. The input data is grouped in blocks, which are sized according to proper heuristics. Historical measurements collected for the same data rate and block interval are fed to the Isotonic Regression[18] algorithm to calculate the optimal batch interval. There are three parameters in the algorithm: a constant value used to control how frequent are the changes of the batch and block intervals, the block interval incremental step size, and the data injection rate discretization. The DyBBS algorithm has been evaluated in the Spark Streaming framework,[19] which is a batch-based stream processing system for distributed architectures.

GASSER[15] is a new system targeting window-based streaming operators on heterogeneous multicores. It offloads the execution of sliding-window operators on local GPUs by using CUDA. The runtime is based on the FastFlow library.[20] It implements an online learning model to find the optimal value of parallelism (number of CPU threads) and batch size (configuration in the sequel), improving the system throughput and maintaining at the same time the latency as lower as possible. The basic strategy involves generating random configurations (within a given range depending on the number of cores and on the system features of the GPU) by using a low discrepancy generator. Then, these values are tested while the application is running by monitoring the system throughput. Finally, the prediction model is refined by using the measured data until it finds the optimal values that maximize throughput and minimize the latency. Differently from GASSER, our work targets stream-based data compression applications focusing more on the aspect of reducing the end-to-end latency rather than improving the system throughput.

In GStream,[21] the batch size is controlled by an elastic API bounded by two user-defined parameters called minimum (*min*) and maximum (*max*), where the default values are 1 and 4096. The communication between two distinct operators of the work-flow graph is implemented through data queues. The batch size is controlled by the *pop* operation on the input queue of a given operator: if there is less than *min* elements in the queue, the processing stage blocks and waits for the producer to enqueue more data; if there is more than *min* but less than *max* elements in the queue, all elements in the queue are removed to be processed; if there is more than *max* elements in the queue, *max* elements are removed for processing.

Both Saber[17] and G-Storm[22] aim to utilize the GPU for parallel stream processing. Saber[17] presents a scheduling strategy to execute streaming window-based SQL queries on both CPU and GPU. G-Storm[22] integrates with Apache Storm to support GPUs using JCUDA. Both of them use batching of data tuples before offloading them to the GPU. However, they do not adapt dynamically their batch size to improve a given latency target.

Table 1 presents a summary of the related work to highlight the main differences among them. Microbatch adaptation in general is provided by Das et al[9] and DyBBS[16] for a distributed stream processing system while G-Storm[22] just implemented GPU support for Apache Storm using JCUDA.

TABLE 1 Related work summary

Reference	Microbatch stream processing approach	GPU support	Goal	Used technologies
Das et al[9]	Microbatch sizes are defined by the time intervals.	No	Adaptive algorithm to find the minimum batch size that is smaller than the batch processing time to ensure the system stability.	Spark streaming[19]
DyBBS[16]	Data are grouped in blocks to be grouped in microbatches, which are defined by the time intervals.	No	Adaptive algorithm to minimize the end-to-end latency by batch and block sizing while ensuring system stability in the Spark Streaming system.	Spark streaming[19]
G-Storm[22]	Microbatch size is defined by the number of tuples.	Yes	GPU support for Apache Storm.	Apache Storm and JCUDA[23]
Saber[17]	Window-based streaming SQL queries.	Yes	A hybrid stream processing engine for heterogeneous architectures.	Java
GStream[21]	Microbatch sizes are defined by the speed of the previous pipeline stage and bounded by user-defined parameters.	Yes	Provide high-level abstractions for stream processing on GPUs	C++ templates and CUDA
GASSER[15]	Sliding-window operators.	Yes	Adaptive algorithm to find the best concurrency level and batch size for improving throughput (windows processed per second) and minimizing latency.	FastFlow[20] and CUDA
This work	Microbatch sizes are defined by the data length.	Yes	Adaptive algorithms that meet a target latency expressed by the application programmer.	SPar[24,25] and CUDA

Both Saber[17] and GStream[21] systems deal with data stream processing on GPUs without the adaptive microbatching support. Only GASSER[15] proposed the adaption of batch size for stream processing on GPUs, however, the adaptation was not elastic. Furthermore, it is worth noticing that none of the previous research works aimed at meeting a target SLO expressed by the application programmer by using C++ attributes. They followed the classical high-performance approach with the objective to extract the maximum performance possible from the parallel architectures at hand providing the application programmer with a scalable solution.

3 ADAPTIVE MICROBATCHING FOR STREAM PROCESSING ON GPUS

Stream processing applications are characterized by the continuous processing of data coming from one or more input streams. These input streams are usually infinite, where the data are generated continuously by sources such as sensors, financial tickers, or social media. The data input rate usually varies over the time, influenced by several different factors. Historically, programmers write these kinds of applications using smart sequential algorithms. Often, such applications need to be parallelized to fully utilize the resources available on modern heterogeneous architectures and accelerate the application execution to sustain the current input pressure without introducing bottlenecks in the streaming pipeline.

One of the available tools that can be used to quickly parallelize sequential stream processing applications by using high-level parallel abstractions is SPar[3]. SPar is a domain-specific language (DSL) focused on expressing stream parallelism.[24,25] It provides a set of five attributes that can be expressed as C++ annotations to label: (i) a stream parallelism region in the sequential code (using the attribute `ToStream`); (ii) the code of each computing phase (`Stage` attribute); (iii) the data items passed from one stage to another (`Input` and `Output` attributes); (iv) and the degree of parallelism (`Replicate` attribute), which defines the number of replicas of a given `Stage`. The SPar compiler parses these attributes, and then source-to-source transformations are performed to produce parallel code leveraging the FastFlow parallel library.[20]

In our previous work,[5] we identified the need for creating microbatches of stream items to properly exploit many-core accelerators like GPUs with SPar. This raised the problem of properly identifying the microbatch size to improve the performance of stream processing applications. The ideal batch size depends on several characteristics, such as the current workload and input rate. It also depends on the low-level features of the underlying platform. In some cases, a large batch size allows achieving high throughput because of the better exploitation of the available data transfer bandwidth. However, a large batch might significantly increase the end-to-end latency to produce new results.[9] In addition to that, the unbalancing workload may also interfere in the latency time, making useless any attempts to adapt to these changes proactively.

When developing parallel stream processing applications with a high-level framework like SPar, which aims at simplifying parallel programming, it is of foremost importance that these factors are considered to comply with user requirements in terms of latency or throughput. Due to that, we represent the user requirements with a high-level concept called SLO.[2] By using SLO annotations, the SPar application programmer specifies a performance goal in a stream parallelism region (ie, a SPar's `ToStream`). The syntax and semantics of the SPar language were already extended for adding the possibility to express SLOs with standard C++ attributes in the sequential source code.[2] Therefore, we just reuse this definition to focus on the algorithm design for adapting the batch size at run-time. Listing 1 provides an example of our idea and how the user interacts with the underlying adaptive run-time system, which was developed using a control loop strategy driven by reactive algorithms that elastically adapt the microbatch size. The user needs only to specify the target latency value (set-point) as shown in the first line, using the `slo::Latency` attribute. The proposed algorithms in this article are not exposed to end-users since the SPar will generate them at compile time. Furthermore, if the user expresses an unreachable latency set-point, our algorithms apply a best-effort approach by default. The code example in Listing 1 is just a high-level representation of the use-case application that will be further explained in Section 3.1.

```
[[spar::ToStream, slo::Latency(500)]]
while (!EndOfStream){
    batch[size];
    batch = Read(stream);
    [[spar::Stage, spar::Input(batch), spar::Output(batch)]] {
        batch = FindMatch(batch);
    }
```

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

```
    [[spar::Stage, spar::Input(batch)]] {
        Write(Filter(batch));
    }
}
```

Listing 1: `SPar example of a latency-aware SLO fixed to 500 ms.`

3.1 | Use case: The LZSS application

LZSS[26] is a compression algorithm belonging to the Lempel-Ziv family, which has been used in many compression applications like RAR and PkZip.[27] In this article, we use the parallel implementation on GPU presented in Stein et al.[8] This LZSS version exploits GPU parallelism by using CUDA, and leverages SPar to target CPU cores and to coordinate the different parts of the algorithm. The LZSS application is a valuable candidate because it represents a notable stream processing application. It has important characteristics such as pipeline parallelism on CPU, data parallelism on GPU, data streaming, and unpredictable workload variations. Our strategies and algorithms can be applied in other stream processing applications that present the same pipeline computation/workflow characteristics, such as Dedup and Bzip2 compression algorithms. However, in this article, we focus on this important use case in detail.

The compression phase uses previous elaborated data as a dictionary to find similar data occurrences. For each byte, the algorithm searches in the previous bytes the longest occurrences available and writes in the resulting file the references to these occurrences instead of the bytes themselves, thus reducing the total size of the file. The LZSS algorithm limits the size of the bytes that it will search by using a specific window size. In our case the window size is 4096 bytes. The higher the window size the higher the space to write compressed data. We choose this value because it is a balance between different workloads, which can be more or less compressed. With this size we also are able to save the compression index using only 12 bits in order to optimize the storage space. Furthermore, to preserve data consistency, the last window of the previous microbatch is always concatenated to the next microbatch, because it will be used by the first bytes in the next microbatch for the match operation. In the GPU LZSS version, although the microbatch size of the data items can vary, the result will be always consistent with the original serial version. In both Figure 1 and Listing 1, we sketch the high-level workflow of the LZSS algorithm. It is a three-staged pipeline whose stages can be described as follow:

- **Read**: it reads the input file and creates microbatches. Each microbatch consists of the last 4096 bytes from the last batch and the next bytes to be processed (of a variable size to be properly tuned and adapted);
- **FindMatch**: in this stage the microbatch is transferred to the GPU and the operations to find the longest match within the window is performed for each byte. Each GPU thread runs the operation for one byte. The results are then transferred back to CPU and sent to the next stage;
- **Filtering + Write**: the last stage takes the results produced by the previous stage for each byte and then creates the resulting file.

The FindMatch stage is the most expensive and may have different execution times depending on the characteristics of the input data. Each byte has to be searched in a specific block of data. The more the data can be compressed, the more time the GPU will take to process the whole batch.

3.2 | Closed-loop control strategies

Stream parallel applications are largely implemented employing the pipeline pattern. The closed-loop (also referred as *feedback*) control strategy[28] adopted by SPar to adapt the internal configuration, acts according to the MAPE loop (*Monitor*, *Analyze*, *Plan*, and *Execute*) shown in Figure 1. Such closed-loop strategy has been introduced in the SPar DSL by Griebler et al.[2] The last stage of the pipeline monitors specific metrics of the processed streaming items (eg, elapsed latency) and sends them back to the first stage, which is in charge of analyzing the actual measurements to make the correct decisions, that is, by properly adapting the microbatch size to keep track with external variations.

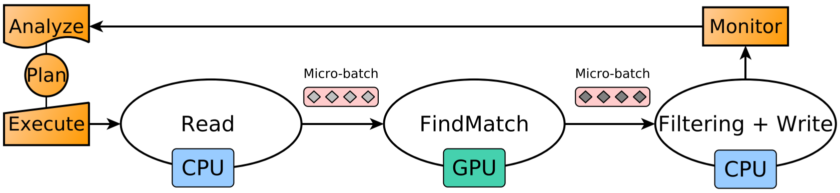

FIGURE 1 The Lempel-Ziv-Storer-Szymanski parallel implementation with the feed-back loop control strategy

GPU-based stream processing applications usually need to leverage dynamic microbatch sizes to process a continuous flow of data. As already hinted before, the size has a direct correlation with the latency of output results. As we will see in Section 4, increasing the microbatch size also increases the latency. Therefore, the closed-loop control strategy implemented has to monitor the current latency obtained by using a given microbatch size. If the latency requirement provided by the user is not respected, the size is changed according to the planning algorithm. The execution phase is in charge of changing the batch size configuration for the next batch of items that have to be computed. This is usually done without blocking the stream to avoid downtimes in the processing. The closed-loop strategy is executed for each new item entering the pipeline to take corrective actions promptly without delays.

We developed four different planning algorithms to make the decision in the control loop named as: fixed adaptation factor (FAF), percentage-based adaptation factor (PBAF), PBAF without threshold (*No Threshold*), and multiplier-based adaptation factor (MBAF). These algorithms are presented in the following subsections. The user always defines the target latency to be obtained, and all the proposed algorithms consider the following two parameters: (a) a threshold, which represents an acceptable percentage of variation in the target latency (this threshold aims to reduce the number of corrections on the microbatch size when the measured latency values are within an acceptable range); and (b) an adaptation factor, which defines the granularity of the adapting operations of each *Plan* phase. All designed algorithms compare the current latency with the user target latency set-point plus the acceptable threshold parameter in order to either increase or decrease the microbatch size accordingly. A decrease happens when the latency crosses the upper-bound threshold and an increase when the latency value becomes smaller than the lower-bound threshold. In Section 4, we evaluate the performance for different combinations of these parameters so that we can find the best configurations for our use case.

3.2.1 | Fixed adaptation factor (FAF)

The FAF algorithm tries to reach the target latency by simply changing the microbatch size in fixed steps until the measured latency crosses the acceptable threshold bounds. When the latency reaches a value higher or lower than the SLO threshold, we use a fixed parameter called *adaptation factor* to increase or decrease the batch size. The Algorithm 1 shows the pseudocode of the FAF planning algorithm. The input for the function is the *LastLatency* , which represents the last latency collected by the monitoring phase. Based on this measured latency, we choose to increase or decrease the batch size observing the thresholds (lines 3 and 6). The microbatch adaptation is applied using fixed steps with a fixed size of the adaptation factor (line 4 and 7).

Algorithm 1. Fixed adaptation factor

```
 1: $UpperLim \leftarrow Target * (1 + Threshold), LowerLim \leftarrow Target * (1 - Threshold)$
 2: procedure PLAN($LastLatency, MicroBatchSize, AdaptationFactor$)
 3:     if $LastLatency > UpperLim$ then
 4:         return $MicroBatchSize - AdaptationFactor$
 5:     end if
 6:     if $LastLatency < LowerLim$ then
 7:         return $MicroBatchSize + AdaptationFactor$
 8:     end if
 9:     return $MicroBatchSize$
10: end procedure
```

While the latency is a user-defined parameter, which is directly related to the application and the user requirements, the *AdaptationFactor* parameter defines the granularity of the operations. A higher *AdaptationFactor* provides better results when the measured latency is far from the desired range. However, it is hard to make fine-grained modifications. On the other hand, a smaller *AdaptationFactor* applies a fine tuning of the latency but at the expense of a longer time to respond to big workload spikes (*settling time* in the control theory jargon).

3.2.2 | Percentage-based adaptation factor (PBAF)

Since the FAF algorithm (Section 3.2.1) works with a FAF, it may not respond so fast to high workload variations. Therefore, we conceived the PBAF algorithm, which uses a percentage of the adaptation factor based on how far the measured latency is from the target.

In this algorithm, the adaptation factor parameter represents the maximum possible adaptation factor. The solution will use the whole adaptation factor when the variation is higher than 60% of the target (*MaxGrowBoundary* variable) and reduce the adaptation factor as the latency reaches a value close to the target. This means that when the measured latency is far from the target, the algorithm behaves just like FAF, but when small adjustments are necessary it uses just a fraction of the adaptation factor to be more precise. Therefore, we expect that the PBAF algorithm will behave better with bigger adaptation factors, which is the worst-case scenario for the FAF algorithm.

The pseudocode of PBAF is reported in Algorithm 2. We divide the measured latency by the target in line 3 to calculate how far we are from the desired set-point. Based on this number, the algorithm calculates an *adaptationPercentage* in line 5 (for latency bigger than the threshold) and line 9 (for latency smaller than the threshold), limited to 100% of the adaptation factor.

Algorithm 2. Percentage-based adaptation factor

```
 1: $UpperLim \leftarrow Target * (1 + Threshold)$, $LowerLim \leftarrow Target * (1 - Threshold)$, $MaxGrowBoundary \leftarrow 0.6$
 2: procedure PLAN($LastLatency, MicroBatchSize, AdaptationFactor$)
 3:     $Perc \leftarrow LastLatency / Target$
 4:     if $LastLatency > UpperLim$ then
 5:         $AdaptationPercentage \leftarrow min((Perc - 1)/MaxGrowBoundary, 1)$
 6:         return $MicroBatchSize - AdaptationFactor * AdaptationPercentage$
 7:     end if
 8:     if $LastLatency < LowerLim$ then
 9:         $AdaptationPercentage \leftarrow min((1 - MaxGrowBoundary)/Perc, 1)$
10:         return $MicroBatchSize + AdaptationFactor * AdaptationPercentage$
11:     end if
12:     return $MicroBatchSize$
13: end procedure
```

3.2.3 | PBAF without threshold

The FAF and PBAF algorithms do not change the microbatch size unless the latency is outside the threshold bounds. This means that the latency tends to stay near the higher or lower bounds of the threshold for both the strategies. Since the algorithm for changing the batch size is not particularly expensive, another possible approaches could try to track the target latency value more precisely, even when the latency measurements are within the desired thresholds. This algorithm, which actually ignores the threshold values, is presented in Algorithm 3. Differently from the two previous approaches, this strategy aims to react better to small unbalances in the workload, changing the microbatch size before the SLO is violated (so acting more proactively in this sense).

Algorithm 3. PBAF without threshold

```
 1: $MaxGrowBoundary \leftarrow 0.6$
 2: procedure PLAN($LastLatency, MicroBatchSize, AdaptationFactor$)
 3:     $Perc \leftarrow LastLatency / Target$
 4:     if $LastLatency > Target$ then
 5:         $AdaptationPercentage \leftarrow min((Perc - 1)/MaxGrowBoundary, 1)$
 6:         return $MicroBatchSize - AdaptationFactor * AdaptationPercentage$
 7:     end if
 8:     if $LastLatency < Target$ then
 9:         $AdaptationPercentage \leftarrow min((1 - MaxGrowBoundary)/Perc, 1)$
10:         return $MicroBatchSize + AdaptationFactor * AdaptationPercentage$
11:     end if
12:     return $MicroBatchSize$
13: end procedure
```

Instead of testing the measured latency against the upper and lower bounds, in lines 4 and 8 of Algorithm 3 we compare it directly to the target latency. The rest of the algorithm is the same as in the PBAF one.

This algorithm keeps trying to improve the latency even if it is already inside the threshold (SLO hit). This can cause fluctuations in the measured latency, which can be seen as a drawback. This might happen because the microbatch size is continuously adjusted to try to match the target. To avoid this behavior, we used a PBAF.

3.2.4 | Multiplier-based adaptation factor (MBAF)

Finding the right adaptation factor for adjusting the batch size dynamically and reactively is challenging in general. In all the previous algorithms, a small adaptation factor limits the capability of the algorithm to adapt to sudden changes in the workload. Aiming at accelerating the convergence of the microbatch size to match the target latency, we propose the MBAF planning algorithm.

In this algorithm, the goal is to converge to the desired latency as fast as possible. To this end, when the value is higher than the upper bound, we use the formula $Target/LastLatency{*}AdaptationFactor$. When it is lower, we change the formula to $(Target + Target - LastLatency)/Target$. The $LastLatency$ is the last latency collected and $Target$ is the target latency expressed by the application programmer by using the `slo::Latency` attribute. By using these formulas, we can converge to the target latency faster because when the latency is far from the target, this algorithm produces a larger adaptation factor.

In Algorithm 4 is reported the pseudocode for MBAF, where lines 4 and 8 represents the formula to generate a multiplier for the adaptation factor. In lines 5 and 9, the *AdaptationMultiplier* generated by the formula is multiplied by the adaptation factor.

Algorithm 4. Multiplier-based adaptation factor

```
1:  $UpperLim \leftarrow Target * (1 + Threshold), LowerLim \leftarrow Target * (1 - Threshold)$
2:  procedure PLAN($LastLatency, MicroBatchSize, AdaptationFactor$)
3:      if $LastLatency > UpperLim$ then
4:          $AdaptationMultiplier \leftarrow Target/LastLatency$
5:          return $MicroBatchSize - AdaptationFactor * AdaptationMultiplier$
6:      end if
7:      if $LastLatency < LowerLim$ then
8:          $AdaptationMultiplier \leftarrow (Target + Target - LastLatency)/Target$
9:          return $MicroBatchSize + AdaptationFactor * AdaptationMultiplier$
10:     end if
11:     return $MicroBatchSize$
12: end procedure
```

4 | EXPERIMENTS

We executed the experiments with all the algorithms described in Section 3 by using the LZSS compression application and its GPU parallelization already proposed in Stein et al.[8] All the experiments were carried out on a single machine with a CPU Intel(R) Core(TM) I9-7900X @ 3.3 GHz (10 cores and 20 threads), 32 GB of RAM memory and a Titan XP GPU with compute capability 6.1 and 12 GB 2400MHz of memory. The system was running on Ubuntu OS (kernel 4.15.0-43-generic). All programs have been compiled using -O3 compiler flags. The software we used were G++ 7.3 and NVCC 10.0.130 compiler and SPar.

4.1 | Workloads

We tested the algorithms with four datasets as input to the processing pipeline:

- **Enwiki**[bulleted]: it is a dump of the Wikipedia website in English having a size of about 14 GB;

[bulleted] Available in https://dumps.wikimedia.org/enwiki/20190701/enwiki-20190701-pages-meta-history1.xml-p30017p30303.bz2

- **Custom**: it is a custom dataset that we created to have a specific behavior with spikes, having a size of about 1.6 GB. This was designed by generating random alphabetic characters with repeated file sections that combine 1 to 9 identical data slots. The goal was to simulate workload peaks in the application since LZSS requires more computational work when there are identical data slots (compressible data);
- **Linux**[bulleted]: it is a tar image of the Linux source code having a size of about 816 MB. To have a longer processing time, we repeated this dataset eleven times;
- **Silesia**[bulleted]: it is a corpus of data that represents real-world files (XML, DLLs, and many others) having a size of about 202 MB. We repeated this dataset eleven times to increase the load.

We first ran the application with static batch sizes to analyze the latency variations for each workload. Figure 2 shows the latency results of each workload over the time by using static microbatch sizes of 4, 8, and 12 MB. We noticed that both the EnWiki and Custom datasets have a regular behavior, with some spikes of relative small duration. The Linux dataset takes about 30 seconds to be processed (in this experiment, the original dataset is repeated 11 times). It has two consecutive big spikes at around one-third of the processing time, both lasting about 3 seconds. Finally, Silesia represents a challenging dataset to deal with dynamically adaptation on the batch size to optimize latency. It takes only 10 seconds to be processed, it has a medium and a big spike both lasting about 2.5 seconds.

4.2 | Behavior of the algorithms

This section aims at presenting the differences in the algorithm behavior with the same configuration parameters. In addition to the target latency expressed by the application programmer and the threshold values, the algorithms also have the adaptation factor parameter. However, each algorithm performs differently under different adaptation factors. Some of them perform better with small values while others perform better with big values of the parameters.

To understand the behavior of each algorithm, we chose EnWiki as it is a real dataset while the other three have been somehow customized (ie, by repeating the original dataset). Figure 3 shows the results obtained with EnWiki for 1 second latency SLO and 5% threshold using an adaptation factor of 128 kB. The latency measured during the experiments is plotted as a solid blue line. The target latency is plotted as a solid black line and the upper and lower bounds of the latency are plotted as dashed orange lines. The evolution of the batch size in MegaBytes (MB) is reported as a dotted green line and it is related to the right *Y*-axis.

The FAF algorithm, discussed in Section 3.2.1, remains mostly stable during the execution. However, it has some SLO violations after having converged to the target latency. This is mainly due to the FAF, which forces the algorithm to apply large variations of the microbatch size. The first plot from the right-hand side presents the results of the PBAF algorithm, which we discussed in Section 3.2.2. As expected, the initial convergence phase is equal to the previous algorithm, as it uses 100% of the adaptation factor until it converges to the target latency. The SLO violations are avoided until the latency decline right after the 40th second of execution. After that point, the algorithm takes a long time doing small variations of the batch size to stabilize again within the boundaries. The algorithm would need a bigger adaptation factor to converge faster.

The PBAF without threshold is represented by the second plot from the left-hand side, which shows a behavior similar to the PBAF: the convergence is in general faster when the latency is below the target but takes some time to adapt when the latency is higher than the target. There is also an SLO violation at the 70th second of the execution when the algorithm had just increased the microbatch size to try to reach the target latency because of a workload variation, violating the upper bound threshold. The last plot of Figure 3 shows the behavior of the MBAF algorithm, which represents our best result for the given target latency and threshold. The initial convergence is faster as this algorithm multiplies the adaptation factor according to the distance to the target latency. The latency remains stable except for the decline and the peak halfway of the execution, for which the algorithm presents a quick reaction.

4.3 | Performance of the proposed algorithms

This section aims at presenting a comprehensive overview of the results with the LZSS data compression application. Our goal is to understand how the algorithms work with different configurations and SLOs. We performed experiments using three latency targets (0.5, 1, and

[bulleted] Available in https://www.kernel.org/
[bulleted] Available in http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

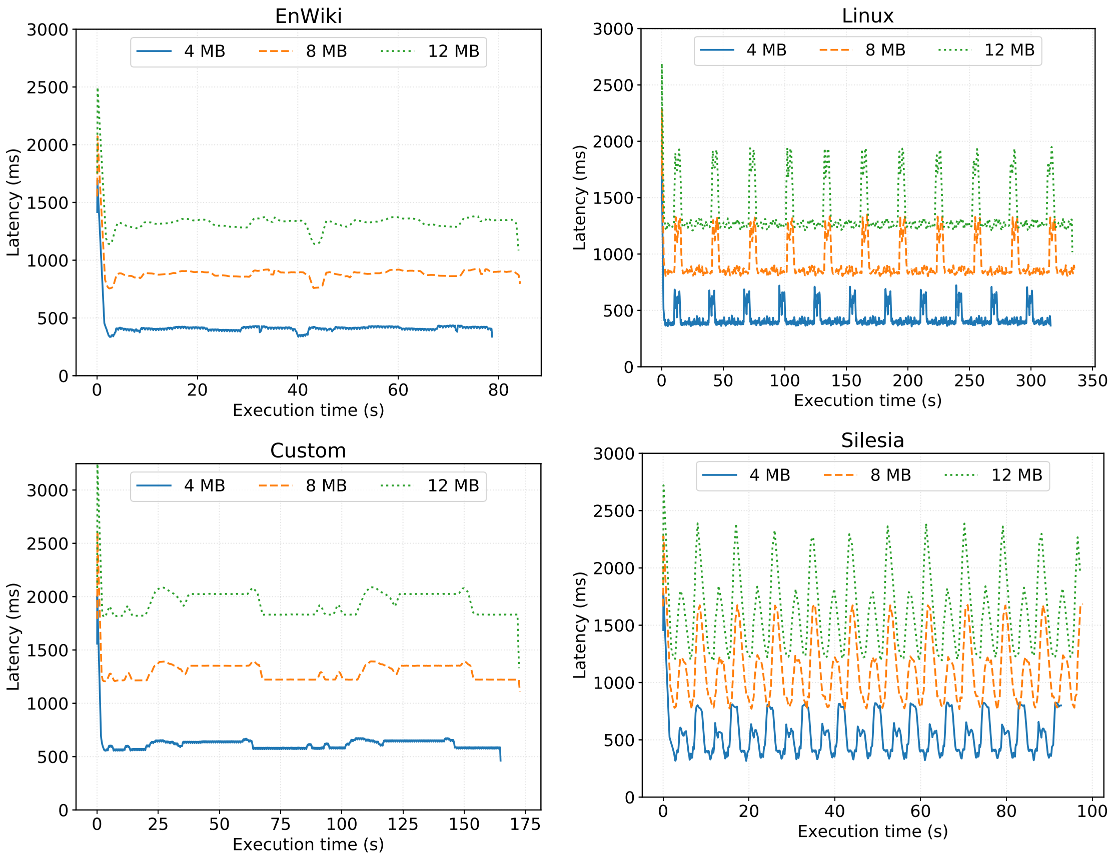

FIGURE 2 Latency variation over time with static microbatch sizes

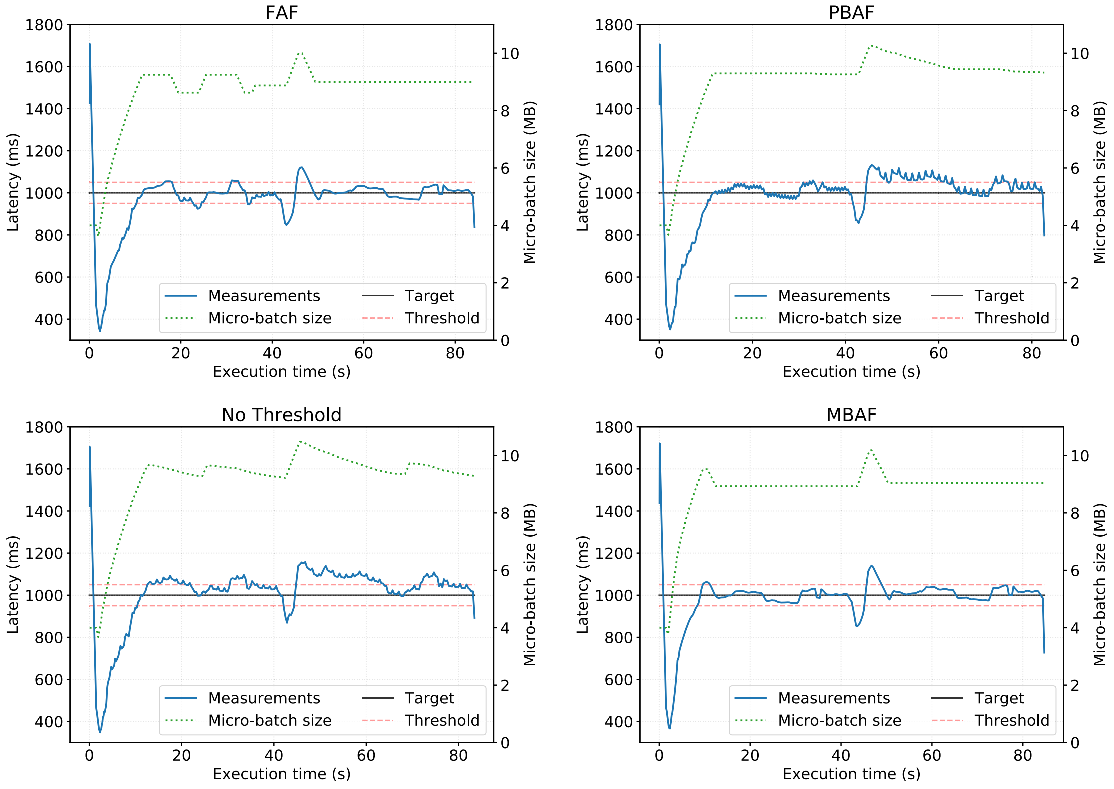

FIGURE 3 Algorithms behavior in EnWiki with target 1 second, threshold 5% and adaptation factor 128 KB

TABLE 2 Best algorithm according to SLO hit (%)

	Custom												
Target	0.5 s				1.0 s				1.5 s				
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%	×
FAF	**90.39***	**99.09***	**99.32***	99.39	**78.83***	94.81	96.27	96.60	68.22	**93.65***	92.55	**95.74***	6
PBAF	86.37	98.39	99.15	**99.45***	77.10	95.28	**96.46***	96.59	63.56	89.61	92.59	95.65	2
No Thresh.	71.41	92.10	99.16	99.37	52.72	80.04	94.15	96.63	50.58	76.22	89.22	93.20	0
MBAF	90.22	98.28	99.25	99.30	77.22	**96.01***	96.32	**97.00***	**71.31***	89.49	**94.66***	95.62	4
	EnWiki												
Target	0.5 s				1.0 s				1.5 s				
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%	×
FAF	83.44	**94.40***	95.78	97.15	69.26	**88.93***	88.67	92.47	58.29	83.15	88.07	90.37	2
PBAF	81.62	93.95	95.64	97.15	70.87	81.35	89.77	92.31	53.14	**85.55***	88.07	90.37	1
No Thresh.	58.39	86.89	92.04	95.34	45.68	71.21	83.60	87.25	38.89	58.76	81.56	86.83	0
MBAF	**83.66***	92.66	**96.30***	**97.24***	**76.60***	86.67	**92.03***	**93.02***	**63.54***	80.56	**92.35***	**92.35***	9
	Linux												
Target	0.5 s				1.0 s				1.5 s				
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%	×
FAF	51.64	**79.18***	**88.36***	**90.44***	63.77	68.62	73.04	78.50	59.02	67.39	70.13	81.15	3
PBAF	**63.80***	77.75	85.39	89.71	**67.29***	74.21	77.34	83.23	64.44	72.57	76.72	84.91	2
No Thresh.	44.63	70.16	84.12	86.42	57.11	**77.19***	**81.71***	**84.02***	57.38	**74.57***	**79.75***	82.90	5
MBAF	58.18	77.19	86.56	89.37	67.04	74.13	76.92	82.53	**64.62***	71.56	74.73	**85.92***	2
	Silesia												
Target	0.5 s				1.0 s				1.5 s				
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%	×
FAF	10.41	17.49	25.29	36.33	8.28	18.09	30.16	48.31	15.66	30.81	45.96	57.84	0
PBAF	13.85	24.22	33.13	44.29	14.89	21.13	35.64	48.08	13.66	29.26	**47.15***	60.00	1
No Thresh.	18.12	**32.06***	**47.65***	**54.91***	**19.01***	**30.71***	**41.43***	**53.54***	**20.45***	**35.23***	46.02	54.24	9
MBAF	**18.83***	29.51	35.89	42.58	12.16	20.22	32.59	49.82	12.95	30.53	47.12	**64.25***	2

Abbreviations: FAF, fixed adaptation factor; MBAF, multiplier-based adaptation factor; PBAF, percentage-based adaptation factor; SLO, service level objective.

*Boldface numbers represent best algorithm in that specific set of requirements.

×Number of times the algorithm was the best in that specific set of requirements.

1.5 s), four different thresholds (5%, 10%, 15%, and 20%), and four adaptation factors (64, 128, 256, and 512 kB) to evaluate the impact of these parameters.

In order to summarize the experimental results[5], we calculated the average SLO hit considering all datasets, proposed algorithms, target latency values, thresholds, and adaptation factors. We define the SLO hit as the percentage of processed microbatches that fall within the threshold bounds. A summary of the best SLO hit we achieved is also presented in Table 2 as well as the best performing adaptation factor for each algorithm in Table 3. Figures 4 to 6 present the SLO hit for each algorithm with targets 0.5, 1, and 1.5 seconds, respectively. The three-dimensional (3D) plots present the impact of the different adaptation factors and thresholds on each target latency and algorithm. As expected, the higher the thresholds the higher the SLO hit rates.

For the target 0.5 second, presented in Figure 4, the algorithms obtained better results with smaller adaptation factors. This is due to the fastest convergence at the beginning of the execution, thus requiring only small changes in the batch size to keep the latency within the SLO boundaries.

[5] We evaluated all the combinations of the parameters, which generated 768 experiments in the total.

TABLE 3 Adaptation factor on best algorithm versions (kB)

	Custom											
Target	0.5 s				1.0 s				1.5 s			
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%
FAF	**64***	**128***	**128***	128	**128***	128	256	256	128	**512***	256	**512***
PBAF	256	128	128	**256***	128	512	**256***	256	256	512	512	512
No Thresh.	64	64	64	64	128	64	128	256	128	128	256	256
MBAF	64	128	256	256	64	**256***	128	**256***	**128***	128	**512***	256
	EnWiki											
Target	0.5 s				1.0 s				1.5 s			
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%
FAF	64	**64***	128	256	128	**256***	256	512	256	512	512	512
PBAF	256	64	128	256	512	256	256	512	512	**512***	512	512
No Thresh.	128	128	64	64	512	128	256	256	256	128	256	512
MBAF	**64***	64	**64***	**128***	**128***	256	**256***	**256***	**256***	512	**512***	**512***
	Linux											
Target	0.5 s				1.0 s				1.5 s			
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%
FAF	64	**128***	**256***	**256***	128	256	256	64	128	512	256	128
PBAF	**64***	64	256	256	**128***	128	128	64	128	256	256	256
No Thresh.	64	64	64	64	64	**64***	**128***	**256***	128	**256***	**256***	512
MBAF	64	64	256	256	128	128	128	64	**256***	256	128	**128***
	Silesia											
Target	0.5 s				1.0 s				1.5 s			
Threshold	5%	10%	15%	20%	5%	10%	15%	20%	5%	10%	15%	20%
FAF	128	128	64	64	64	64	64	128	512	512	512	512
PBAF	64	64	64	64	256	256	256	128	512	512	**512***	512
No Thresh.	64	**64***	**64***	**64***	**128***	**256***	**256***	**256***	**512***	**512***	512	512
MBAF	**128***	64	64	64	256	128	128	128	256	512	512	**512***

Abbreviations: FAF, fixed adaptation factor; MBAF, multiplier-based adaptation factor; PBAF, percentage-based adaptation factor.
* Boldface numbers represent best algorithm in that specific set of requirements.

MBAF was the best algorithm for the thresholds 5% and 10% (with 62.52% and 74.37% SLO hit, respectively), however, the No Threshold algorithm performed better with thresholds 15% and 20% (with 80.74% and 84.01% SLO hit, respectively). All these results have been obtained using an adaptation factor of 64 kB.

In Figure 5, we present the results of the experiments targeting 1 second latency. The MBAF algorithm outperforms all other ones for 5% and 10% thresholds (with 57.53% and 68.72% SLO hit, respectively). The No Threshold and PBAF algorithms show the best results for 15% threshold (with 74.79% SLO hit using an adaptation factor of 128 kB and 74.78% SLO hit using an adaptation factor of 256 KB, respectively). The No Threshold algorithm also presents the best result for the 20% threshold (with 80.36% SLO hit using an adaptation factor of 256 kB).

The last target latency tested was 1.5 seconds, whose results are presented in Figure 6. All algorithms have to spend a significant amount of time in the initial convergence phase to reach the target latency expressed by the application programmer. Therefore, bigger adaptation factors provide better overall results. Notable exceptions are MBAF and FAF with 5% threshold and an adaptation factor of 512 kB, which present a low SLO hit. This occurs because with such a narrow target and big adaptation factor these algorithms are unable to perform the small microbatch size adjustments required to stay within the threshold bounds. Nevertheless, MBAF was still the best algorithm for the 5% threshold, with 51.92% SLO

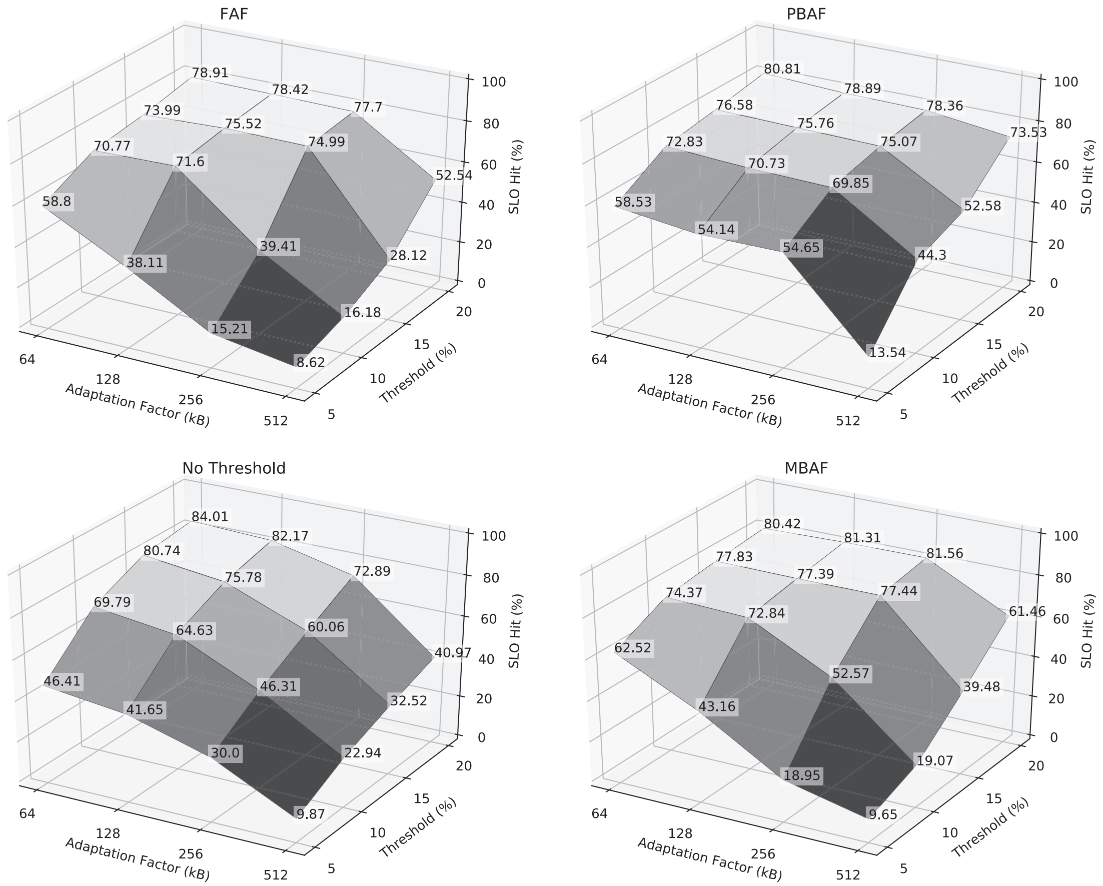

FIGURE 4 Average service level objective hit among all workloads targeting 0.5 second latency

hit using 128 kB adaptation factor. For 10% threshold, PBAF and FAF provided the best results both using an adaptation factor of 512 kB (68.76% and 68.75% SLO hit, respectively). MBAF provided the best results for 15% and 20% threshold and an adaptation factor of 512 kB (with 77.13% and 82.65% SLO hit, respectively).

In the Custom workload (described in Section 4.1) the FAF, PBAF, and MBAF algorithms had very similar results. However, FAF had better overall results, mostly in 0.5 second target because of the smaller initial time to converge. The MBAF algorithm is dominating in the EnWiki workload. Only three exceptions were found, where FAF algorithm was better in two cases and the PBAF algorithm was better in one, but with very narrow margins. In the Linux workload, the No Threshold algorithm provides better results for bigger targets and thresholds, while FAF provides better results for the 0.5 second target. PBAF presented the best results for the 5% threshold experiments for both 0.5 and 1.0 second targets, with very small differences with MBAF in the 1.5 seconds target. Finally, the No Threshold algorithm provides the best overall results for the Silesia workload, which is the most challenging one. In fact, only half of the microbatches proposed by the algorithm reached the SLO.

In summary, we can highlight that MBAF is the best algorithm for the closed-loop control strategy when the input datasets feature low workload fluctuations at run-time. The algorithm can quickly converge to the target SLO and it is also able to adapt the microbatch size when small changes in the workload occur. For workloads with higher fluctuations, the No Threshold algorithm provided, on average, the best results among all the proposed algorithms.

4.4 | Impact of the workloads

In the previous section, we analyzed the average SLO hit among all the workloads for each target latency. In this section instead, we will analyze the impact of the workloads in the SLO hit to understand how the algorithms perform with different workloads. Figures 7 to 10 present the average SLO hit among all the targets for each algorithm with threshold 5%, 10%, 15%, and 20%, respectively. The 3D plots present the impact of the different adaptation factors and workloads on each threshold configuration and algorithm.

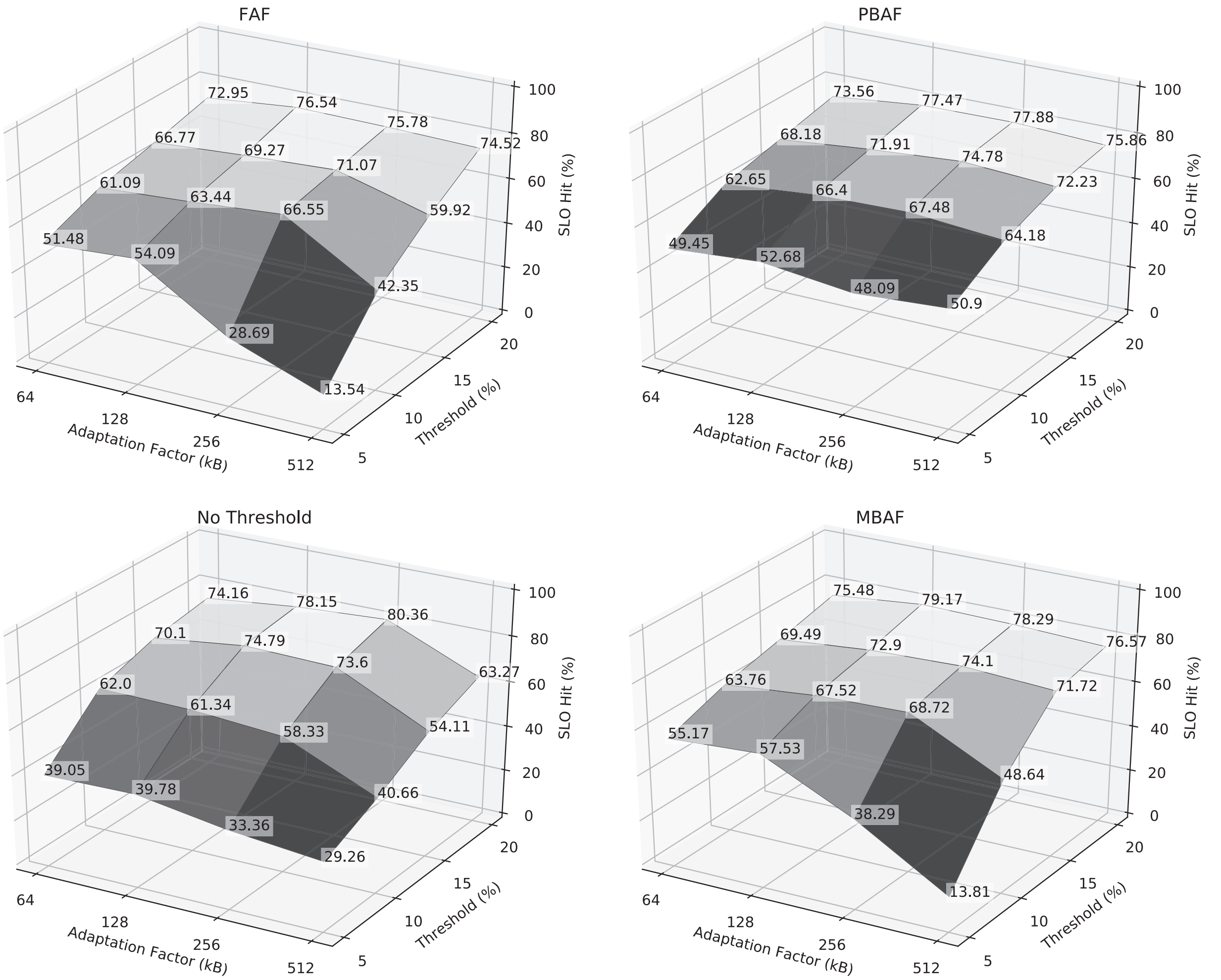

FIGURE 5 Average service level objective hit among all workloads targeting 1 second latency

Figure 7 presents the results obtained using a threshold of 5%. Smaller adaptation factors obtain better SLO hit in general, which can be explained by the small tweaks that are necessary to keep the latency within the tighter bounds. For the Custom workload, the FAF and MBAF algorithms with an adaptation factor of 64 kB obtain the best results (78.58% and 78.01%, respectively). The MBAF with an adaptation factor of 128 kB has the best SLO hit (70.25%) for the EnWiki workload. For the Linux workload, the PBAF algorithm obtains the best SLO hit (62.36%) using an adaptation factor of 128 kB. The best SLO hit (14.33%) for the Silesia workload with 5% threshold is obtained by the No Threshold algorithm using an adaptation factor of 512 kB.

Figure 8 shows the SLO hit rates obtained using 10% threshold. The MBAF algorithm has the best SLO hit rates for the Custom (94.08% using an adaptation factor of 128 kB) and EnWiki (84.02% using an adaptation factor of 256 kB) workloads. However, for the Linux workload the PBAF algorithm performs better (74.42% of SLO hit) with an adaptation factor of 256 kB. For the Silesia workload, the best SLO hit (25.7%) is obtained by the No Threshold algorithm with an adaptation factor of 256 kB.

For the 15% threshold experiment, which is presented in Figure 9, the MBAF algorithm achieves the best SLO hit rate (96.62%) in the Custom workload, using an adaptation factor of 256 kB. For the EnWiki and Linux workloads, PBAF obtains better results (89.82% and 79.79%, respectively), using 512 and 256 kB, respectively. For the Silesia workload, the No Threshold algorithm has the best SLO hit (42.05%) using an adaptation factor of 128 kB.

Figure 10 presents the results of the experiments using a threshold of 20%. The MBAF algorithm has the best SLO hit for the Custom (97.3% with an adaptation factor of 256 kB) and Linux (85.03% with an adaptation factor of 128 kB) workloads. For the EnWiki workload, the PBAF and MBAF algorithms have the best SLO hit (92.26% and 92%, respectively), both with an adaptation factor of 512 kB. Finally, for the Silesia workload, the No Threshold algorithm again obtains the best SLO hit (52.62%), with an adaptation factor of 256 kB.

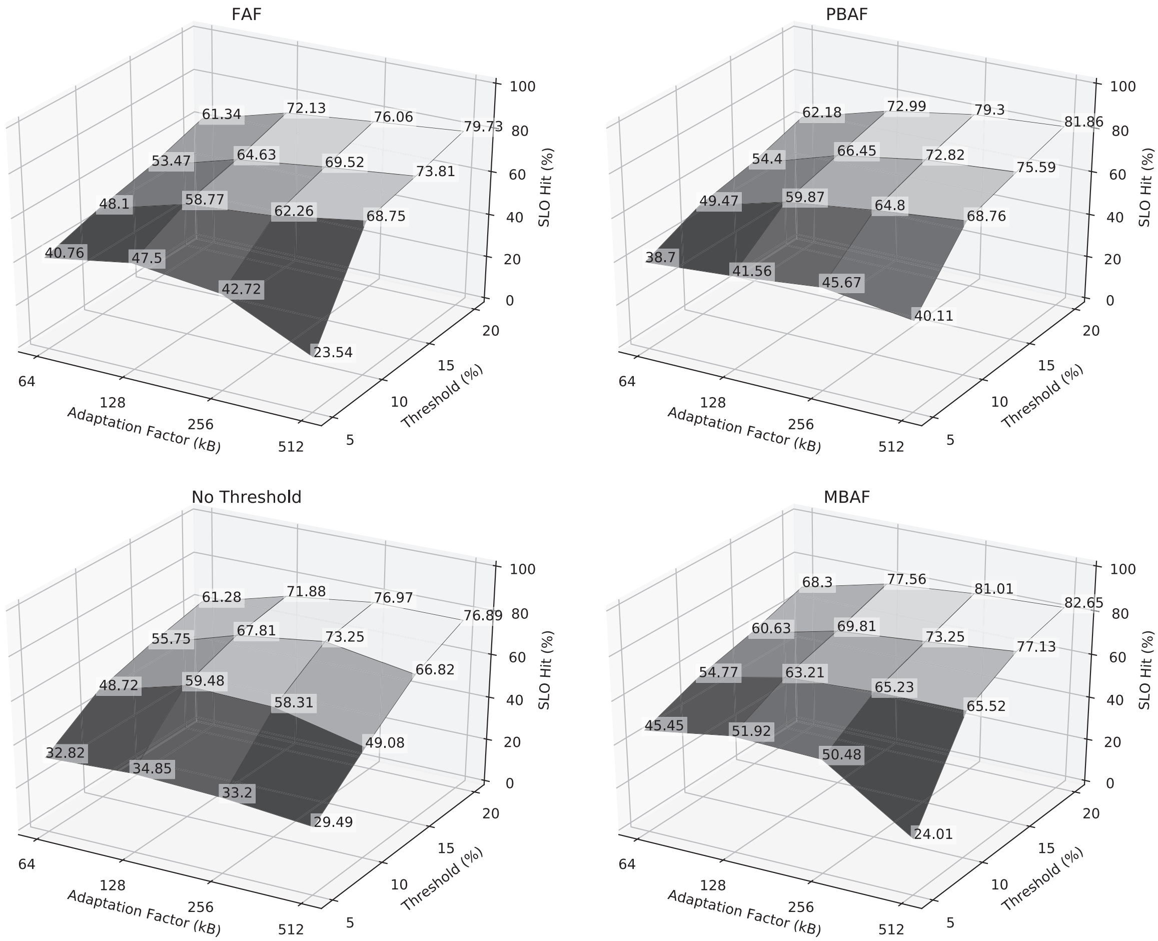

FIGURE 6 Average service level objective hit among all workloads targeting 1.5 second latency

In general, the plots show that higher thresholds seem to favor higher adaptation factors. This is partly explained due to the lower probability of an overreaction of the algorithm, given that even if the reaction exceeds the target latency the measured latency is still inside the threshold. It is also worth noting that Silesia is the most challenging dataset, as previously discussed, which can be perceived by the lower SLO hit in Figures 7 to 10. Nevertheless, the No Threshold algorithm obtains the best SLO hit for this workload with all the threshold values. This can be explained by the highly unstable nature of the workload, combined with the highly active nature of the algorithm.

As already pointed out, each algorithm behaves differently with respect to the adaptation factor. However, the best adaptation factor also depends on the target latency (provided by the user) and the acceptable threshold percentage. Therefore, in Table 2, we classified the experimental results according to the workload, target latency, and threshold. The table presents the best SLO hit for each algorithm, using the best adaptation factor for that SLO. The best algorithm for the workload and the SLO is shown in boldface and marked with an asterisk. Finally, we present the adaptation factor used in all these versions in Table 3. In summary, FAF was the best algorithm in 11 cases (most of it in Custom workload), PBAF was the best algorithm in six cases, No Threshold was the best algorithm in 14 cases (most of it in Silesia workload), and MBAF was the best algorithm in 17 cases (most of it in EnWiki workload).

5 | CONCLUSION

This article proposed four novel algorithms for a closed-loop control strategy that tries to meet a given target latency through the dynamic adaptation of microbatches offloaded to GPUs. Our solution is integrated into a high-level parallel programming abstraction for stream parallelism called SPar, where the user can easily express in the source code a target latency using standard C++ attributes. The experiments were carried out with a real-world and representative streamed data compression application (LZSS). The main advantage of our solution is that it tries to use only the necessary computing resources to meet a target latency by adapting the size of the microbatches. Furthermore, to the best of our knowledge,

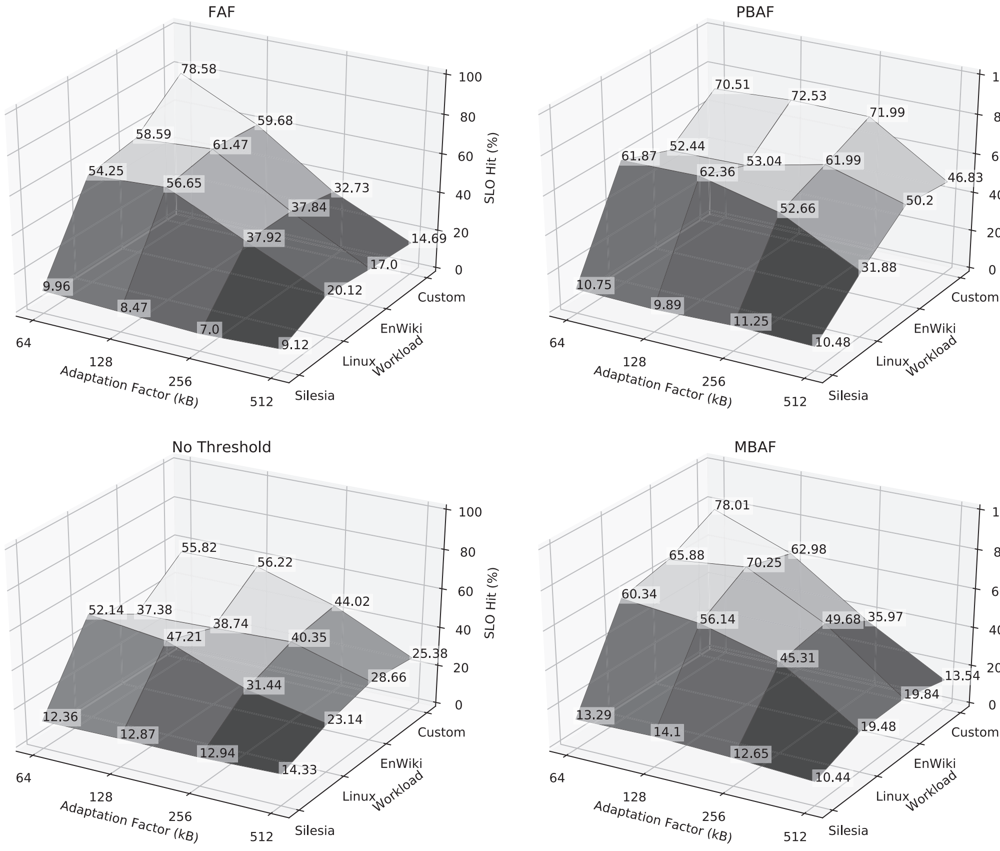

FIGURE 7 Average service level objective hit among all targets with 5% threshold

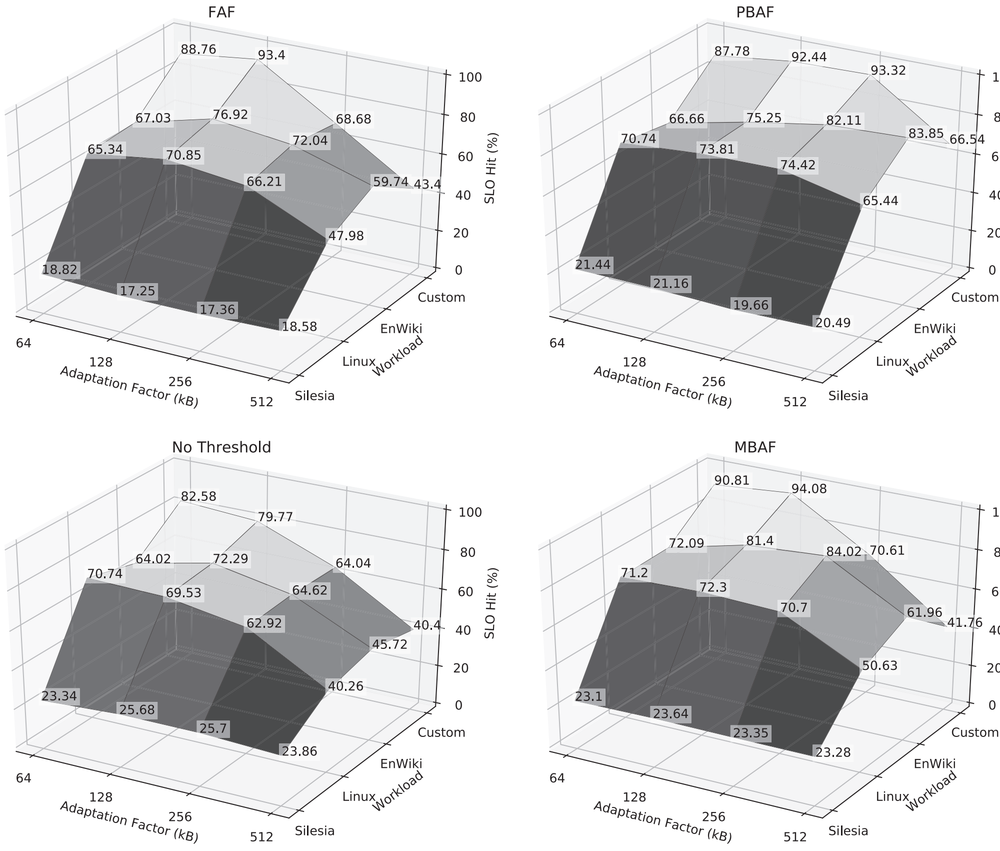

FIGURE 8 Average service level objective hit among all targets with 10% threshold

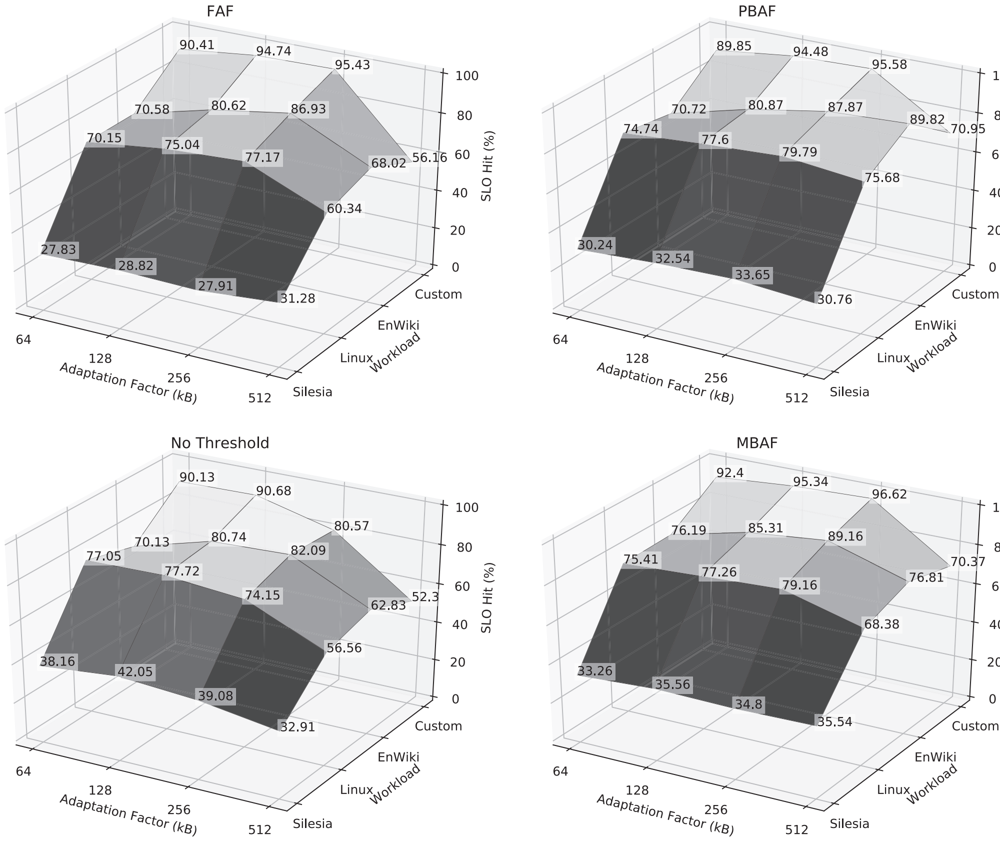

FIGURE 9 Average service level objective hit among all targets with 15% threshold

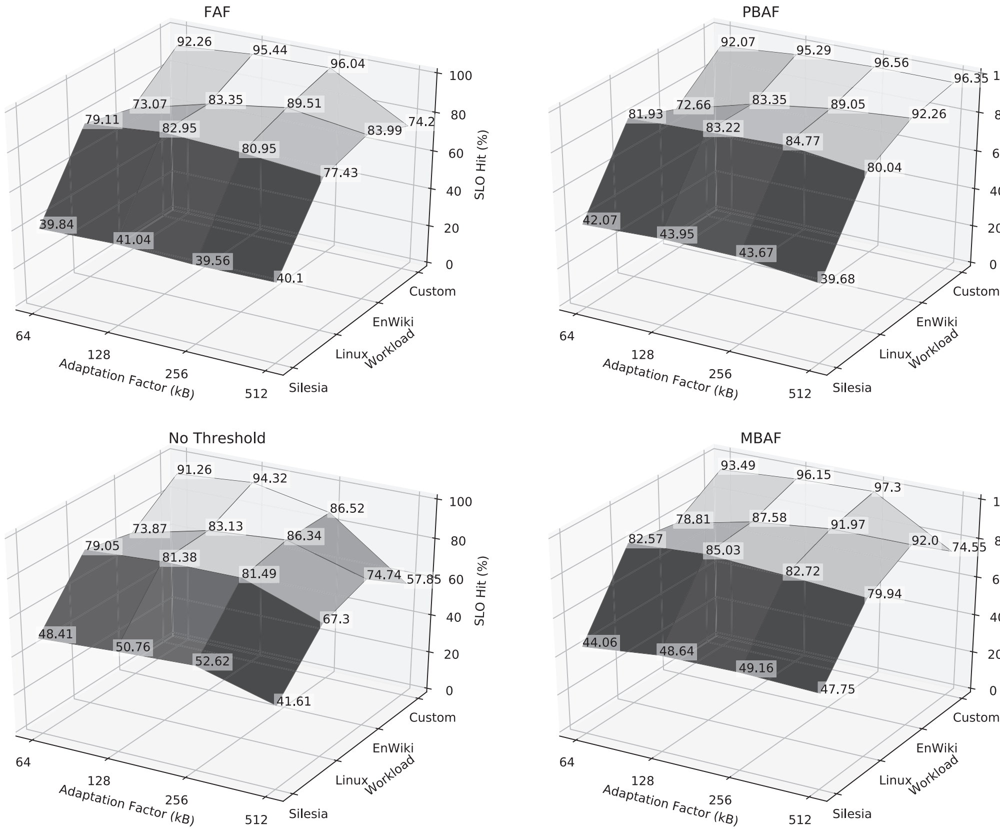

FIGURE 10 Average service level objective hit among all targets with 20% threshold

our solution is the only one able to adapt elastically and reactively at run-time so that it can respond to unpredictable workload fluctuations. We observed a trend that algorithms with elastic adaptation factor respond better for more stable workloads, while algorithms with narrower targets respond better for highly unbalanced workloads.

Although we tested the algorithm on experiments using a real-world streamed data compression application, our performance results cannot be generalized to all stream processing applications. These experiments also depend on the CPU and GPU architectures as well as on the Operating System. As future work, it is possible to evaluate our strategy and algorithms on other stream processing applications, as well as testing using other frameworks like OpenCL. Our strategy and algorithms could also be implemented and tested for a multi-GPU environment. Finally, we believe that there is space for improving the current strategy and algorithms by creating new strategies and also triggering different adaptation techniques based on specific behavior of the application.

ACKNOWLEDGMENTS

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 thanks the Laboratory of Advanced Research on Cloud Computing (LARCC)[6] for the computing resources support to this research work.

ORCID

Dinei A. Rockenbach https://orcid.org/0000-0002-2091-9626
Dalvan Griebler https://orcid.org/0000-0002-4690-3964
Massimo Torquati https://orcid.org/0000-0001-6323-3459
Gabriele Mencagli https://orcid.org/0000-0002-6263-7723
Marco Danelutto http://orcid.org/0000-0002-7433-376X
Luiz G. Fernandes https://orcid.org/0000-0002-7506-3685

REFERENCES

1. Azara J, Makhoula A, Barhamgib M, Couturiera R. An energy efficient IoT data compression approach for edge machine learning. *Future Generat Comput Syst*. 2019;96:168-175. https://doi.org/10.1016/j.future.2019.02.005.
2. Griebler D, Vogel A, De Sensi D, Danelutto M, Fernandes LG. Simplifying and implementing service level objectives for stream parallelism. *J Supercomput*. 2019;1-26. https://doi.org/10.1007/s11227-019-02914-6.
3. Rathore MM, Son H, Ahmad A, Paul A. Real-time video processing for traffic control in smart city using Hadoop ecosystem with GPUs. *Soft Comput*. 2018;22(5):1533-1544. https://doi.org/10.1007/s00500-017-2942-7.
4. Yang S. IoT stream processing and analytics in the Fog. *IEEE Commun Mag*. 2017;55(8):21-27. https://doi.org/10.1109/MCOM.2017.1600840.
5. Rockenbach DA, Stein CM, Griebler D, et al. stream processing on multi-cores with GPUs: parallel programming models' challenges. Paper presented at: Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2019:834-841; IEEE; Rio de Janeiro, Brazil.
6. Patel RA, Zhang Y, Mak J, Davidson A, Owens JD. Parallel lossless data compression on the GPU; 2012:1-9; IEEE.
7. Zu Y, Hua B. GLZSS: LZSS Lossless Data Compression Can Be Faster. Paper presented at: Proceedings of Workshop on General Purpose Processing Using GPUs; 2014:46:46-46:53; ACM, Salt Lake City, UT.
8. Stein CM, Dalvan G, Marco D, Gustavo FL. Stream parallelism on the LZSS data compression application for multi-cores with GPUs. Paper presented at: Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2019:247-251; IEEE, Pavia, Italy.
9. Tathagata D, Yuan Z, Ion S, Scott S. Adaptive stream processing using dynamic batch sizing. Paper presented at: Proceedings of the Adaptive Stream Processing Using Dynamic Batch SizingSOCC; 2014:16:1-16:13; ACM, Seattle, WA.
10. Vogel Adriano, Griebler Dalvan, Sensi Daniele De, Danelutto Marco, Fernandes Luiz Gustavo. Autonomic and latency-aware degree of parallelism management in SPar. Paper presented at: Proceedings of the European Conference on Parallel Processing.Lecture Notes in Computer Science; 2018:28-39; Springer, Turin, Italy.
11. De Sensi D, Torquati M, Danelutto M. A reconfiguration algorithm for power-aware parallel applications. *ACM Trans Architec Code Optimiz*. 2016;13(4):43:1-43:25. https://doi.org/10.1145/3004054.
12. De Matteis T, Mencagli G. Keep Calm and React with Foresight: Strategies for Low-Latency and Energy-Efficient Elastic Data Stream Processing; 2016:13:1-13:12.
13. NVIDIACUDA. C Programming Guide. PG-02829-001_v9.2; 2018.
14. The Khronos Group, The OpenCL Specification Version 2.2-11; 2019.
15. De Matteis T, Mencagli G, De Sensi D, Torquati M, Danelutto M. GASSER: an auto-tunable system for general sliding-window streaming operators on GPUs. *IEEE Access*. 2019;7:48753-48769. https://doi.org/10.1109/ACCESS.2019.2910312.
16. Quan Z, Yang S, Routray Ramani R, Weisong S. Adaptive block and batch sizing for batched stream processing system. *ICAC '16*. Wurzburg, Germany: IEEE; 2016.

[6] LARCC home page: http://larcc.setrem.com.br

17. Koliousis A, Weidlich M, Fernandez RC, Wolf AL, Costa P, Pietzuch P. SABER: window-based hybrid stream processing for heterogeneous architectures. *SIGMOD '16*. San Francisco, CA: ACM; 2016:555-569.
18. Barlow RE, Bartholomew DJ, Bremner JM, Brunk HD. *Statistical Inference Under Order Restrictions: The Theory and Application of Isotonic Regression*. Hoboken, NJ: John Wiley; 1972.
19. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: fault-tolerant streaming computation at scale. *SOSP '13*. New York, NY: ACM; 2013:423-438.
20. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. FastFlow: high-level and efficient streaming on multi-core. In: Pllana S, Xhafa F, eds. *Programming Multi-core and Many-core Computing Systems Parallel and Distributed Computing*. Hoboken, NJ: John Wiley; 2017.
21. Zhang Y, Mueller F. *GStream: A General-Purpose Data Streaming Framework on GPU Clusters*. Taipei City, Taiwan: IEEE; 2011:245-254.
22. Chen Z, Xu J, Tang J, Kwiat KA, Kamhoua CA, Wang C. GPU-accelerated high-throughput online stream data processing. *IEEE Trans Big Data*. 2018;5(2):191-202. https://doi.org/10.1109/TBDATA.2016.2616116.
23. Yan Y, Grossman M, Sarkar V. *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA*. Berlin, Heidelberg: Springer; 2009:887-899.
24. Griebler D, Danelutto M, Torquati M, Fernandes LG. SPar: a DSL for high-level and productive stream parallelism. *Parall Process Lett*. 2017;27(01):1740005. https://doi.org/10.1142/S0129626417400059.
25. Griebler D. Domain-Specific Language & Support Tool for High-Level Stream Parallelism (PhD thesis). Faculdade de Informática-PPGCC-PUCRSPorto Alegre, Brazil; 2016.
26. Michael D. *LZSS (LZ77) Discussion and Implementation*; 2018.
27. Salomon D. *Data Compression: The Complete Reference*. London, UK: Springer; 2007.
28. Hellerstein JL, Diao Y, Parekh S, Tilbury DM. *Feedback Control of Computing Systems*. Hoboken, NY: John Wiley & Sons, Inc.; 2004.

How to cite this article: Stein CM, Rockenbach DA, Griebler D, et al. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurrency Computat Pract Exper*. 2021;33:e5786. https://doi.org/10.1002/cpe.5786