Evaluation of Adaptive Micro-batching Techniques for GPU-accelerated Stream Processing*

Ricardo Leonarczyk[1][0000−0002−5202−5694], Dalvan Griebler[1][0000−0002−4690−3964], Gabriele Mencagli[2][0000−0002−6263−7723], and Marco Danelutto[2][0000−0002−7433/−376X]

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil
[2] Computer Science Department, University of Pisa, Pisa, Italy

Abstract. Stream processing plays a vital role in applications that require continuous, low-latency data processing. Thanks to their extensive parallel processing capabilities and relatively low cost, GPUs are well-suited to scenarios where such applications require substantial computational resources. However, micro-batching becomes essential for efficient GPU computation within stream processing systems. However, finding appropriate batch sizes to maintain an adequate level of service is often challenging, particularly in cases where applications experience fluctuations in input rate and workload. Addressing this challenge requires adjusting the optimal batch size at runtime. This study proposes a methodology for evaluating different self-adaptive micro-batching strategies in a real-world complex streaming application used as a benchmark.

Keywords: Stream processing · Micro-batching · Multicores · GPUs

1 Introduction

Stream processing systems (SPSs) such as Apache Flink[3] and Apache Spark Streaming[4] have traditionally been deployed in commodity clusters, aiming at horizontal scalability [5]. In these scenarios, the emphasis is typically on input/output (I/O), accompanied by relatively low computing requirements that involve logic for data filtering and transformations. However, high-capacity computing infrastructures become critical when there is the need to maintain acceptable *service level objectives* (SLO) (e.g., bounds in terms of latency) for stream processing applications (SPAs) with high computational demands, such as those in computer vision and robotics.

* This research has been supported by the Italian Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing.
[3] https://flink.apache.org
[4] https://spark.apache.org/streaming

Accelerators such as graphics processing units (GPUs) offer significant advantages in such cases due to their capacity for massive data parallelism. However, the integration of a GPU into an SPS can exhibit several challenges. In streaming analytics, data streams convey small items in the form of records of attributes (which we call *tuples*). Such tuples are often small (e.g., a few hundred bytes), so individually processing them on GPU leads to the under-utilization of its computing capacity.

A potential solution to the challenge of GPU integration into SPSs is the processing of stream tuples in micro-batches. Micro-batching is a technique used in SPSs to process small batches of data simultaneously rather than processing tuples individually. The SPS collects a defined number of incoming tuples, groups them into a batch, and processes them as a unit of computation. This is the default processing model for SPSs such as Spark Streaming. In contrast, for tuple-at-a-time SPSs like Flink, the user usually performs micro-batching manually when co-processors are present in the system.

Once micro-batching is adopted, a subsequent challenge arises in determining the optimal batch size. Increasing the batch size in GPU-accelerated SPAs leads to a rise in both buffering requirements and the time it takes for the GPU to start processing. This results in a trade-off between latency and throughput, which can be managed by finding a batch size that can satisfy the particular application SLO [7]. If the workload and input rate are stable, finding a suitable batch size can be a one-time task, which the user may manually find or use auto-tuning approaches. However, SPAs are often subjected to workload and input rate fluctuations, making the suitable batch size an evolving target.

Self-adaptation techniques can be used to determine a suitable batch size at different stages of the SPA's execution, as demonstrated in Sect. 2. Broadly defined, self-adaptation is the capability of a system to autonomously change itself to better respond to its dynamic environment [11]. In practice, a self-adaptive system will collect metrics data about the SPA, and based on them, it will perform adaptation actions (e.g., variating the batch size or the parallelism degree) at runtime to achieve defined SLOs.

A recent paper [9] presented four adaptation algorithms for adapting the batch size to keep the SPA's latency within a threshold around a target latency. Our goal in the current study is to provide a methodology (Sect. 3) consisting of a set of metrics that complement the metric used to compare the algorithms and workloads in this original study, offering a novel way to analyze experiments with SLO-constrained SPAs which use micro-batching techniques. Additionally, (in Sect. 4), we evaluate the adaptive strategies and algorithms with a new complex and real-world application. We believe that an understanding of how the algorithms in [9] behave with an entirely different use case will shed more light on their generalizability and usefulness beyond the basic scenarios they were originally evaluated. The application we selected is the *Military Server Benchmark* [2] (MSB), a SPA designed to exploit data parallelism and highly configurable to reflect workload variations and changes in the input rate. We also consider in the evaluation how latency sampling affects the adaptation. The

original study did not address this aspect, which affects the average of a set of latencies corresponding to the process variable in control theory.

2 Related Work

Stein et al. [9] present a control-loop strategy driven by four algorithms for adapting the batch size to keep a user-defined latency SLO in streaming compression applications targeting GPUs. The proposed algorithms expect a target latency provided by the user. Internally, they also consider a threshold indicating an acceptable percentage of variation in latency, as well as a step size for changing the batch size. The metric used to compare the algorithms and workloads is named *SLO hit*, which is the percentage of batches that meet the SLO in relation to the overall batch count.

De Matteis et al. [5] propose Gasser, an SPS that offloads sliding-window operators on GPUs. Gasser adapts the batch size and parallelism to balance latency and throughput. It calibrates a predictive model at the beginning of execution based on the throughput achieved by different configurations on CPU and GPU. However, Gasser's predictive model does not react effectively to irregular workloads. The Chebyshev distance is used in the study to measure the effectiveness of Gasser's auto-tuning approach.

Das et al. [4] introduce the fixed-point iteration method to find the intersection between batch processing time and batch interval, providing adaptability without specifying a step size. However, this algorithm assumes the existence of a batch interval where the processing rate can keep up with the input rate and may suffer from control-loop delays in cases of sudden workload variations.

Zhang et al. [12] propose the DyBBS algorithm, which uses online historical statistics to adapt batch interval through isotonic regression and block interval through a heuristic approach. DyBBS prioritizes accuracy over convergence time but also assumes the presence of a stable batch interval.

In some studies, the batch size is associated with the efficiency of task schedulers. Venkataraman et al. [10] propose Drizzle, which organizes batches into groups and dynamically adapts the group size using a technique based on the additive-increase/multiplicative-decrease (AIMD) feedback control algorithm. Cheng et al. [3] propose adapting the batch interval through the expert fuzzy control (EFC) technique integrated with A-scheduler, a new Spark Streaming scheduler. They also use a reinforcement learning algorithm to adapt job parallelism based on historic workload variations.

From the studies presented, only [9] and [5] adapt the batch size for GPU-accelerated SPAs, demonstrating that this specific scenario is not currently receiving significant attention from the literature. The remaining studies focus on batch size adaptation for SPAs built upon the Spark DSPS. Besides the traditional latency and throughput metrics, [9] uses the SLO hit, while [5] use the notion of distance from a optimal configuration. Our work extends the SLO-related metrics from [9], and introduces distance-related metrics.

3 Evaluation Methodology

Our methodology consists of a new set of metrics: the *SLO hit* and the *SLO distance* metric. They are used to compare the adaptation algorithms among themselves in terms of quality and effectiveness from different perspectives. Each metric has both a less and a more sensitive version. For the SLO hit, we propose a *batched* and an *itemized* definition. The batched SLO hit is described in Definition 1.

Definition 1 (B-SLH). *Let t be the target SLO, h be a threshold percentage such that $0 < h < 1$, B be the set of batches processed during the whole or part of the application execution, and $\omega : B \to \mathbb{R}$ be a mapping of a batch $b_i \in B$ to its measured performance metric value (e.g., in terms of latency or throughput). The set $B' \subseteq B$ containing the batches which fell within threshold bounds is defined as $B' = \{b \in B \mid t * (1 - h) \leq \omega(b) \leq t * (1 + h)\}$. The batched SLO hit is defined as the percentage of batches that fall within threshold bounds, formally as:*

$$B\text{-}SLH = |B'|/|B| \tag{1}$$

What we refer to as the batched SLO hit was the metric chosen by Stein et al.[9] to evaluate the adaptation algorithms they proposed. This metric is useful to understand the effectiveness of the adaptation algorithms for achieving a defined SLO. However, it presents a notable limitation resulting from the focus on the batch level. Specifically, batches containing large quantities of items will have the same weight as batches containing only a few items. This can produce situations where the adaptation algorithm fails to achieve the SLO for the majority of the items processed by the application, but the resultant SLO hit still remains greater than 50%. The problem described with the batched SLO hit becomes more pronounced as the range of batch sizes is increased. To solve this problem, we propose the itemized SLO hit, formalized in Definition 2. It is fundamentally the same as the batched SLO hit, with the additional consideration of the batch sizes.

Definition 2 (I-SLH). *Let the definition of the sets B and $B' \subseteq B$ be the same as in Definition 1, and let $\sigma : B \to \mathbb{N}$ be the mapping of a batch $b_i \in B$ to its size. The itemized SLO hit is defined as follows:*

$$I\text{-}SLH = \frac{\sum_{b' \in B'} \sigma(b')}{\sum_{b \in B} \sigma(b)} \tag{2}$$

The two SLO hit metrics are binary in the sense that a given batch is either inside or outside threshold bounds. Such metrics can work perfectly for users whose only concern is to know whether the application is meeting the SLO. However, they fail to provide information for the researcher/practitioner who wants to know how much the SLO is being met or not. In the latter case, proper SLO distance metrics can be defined to supplement the SLO hit metrics by providing a measure of how far the batches were from the target SLO.

We propose two distance metrics: the MAD-based SLO distance and the SD-based SLO distance. They are calculated in the same fashion as the population's mean absolute deviation (MAD) and the population's standard deviation (SD). The only difference from the standard statistical forms of MAD and SD is that the target SLO value is used instead of the mean. As the second and final step, we divide the value obtained in the first step by the target SLO value. We found that expressing the values in percentage helps with the interpretability because it is expected that the distance values will not surpass more than one time the target SLO. Furthermore, the percentage format fits naturally with the way the thresholds are specified (as a percentage of the target SLO).

We define below the MAD-based SLO distance in Definition 3 and the SD-based SLO distance in Definition 4.

Definition 3 (MAD-D). *Let t be the target SLO, and let the set B and the function ω be as defined in Definition 1. The MAD-based SLO distance is defined as:*

$$MAD\text{-}D = \frac{\sum_{b \in B}(|t - \omega(b)|)}{|B| \cdot t}. \tag{3}$$

Definition 4 (SD-D). *Let t be the target SLO, and let the set B and the function ω be as defined in Definition 1. The SD-based SLO distance is defined as:*

$$SD\text{-}D = \frac{\sqrt{\sum_{b \in B}(|t - \omega(b)|^2)}}{|B| \cdot t} \tag{4}$$

The MAD-based SLO distance arguably provides the most intuitive results when compared to the SD-based SLO distance. When applied over a single batch, it provides a value that can be directly compared with the threshold. In fact, the batched SLO hit metric can be derived from the MAD-based SLO distance by checking if every batch's distance value is less or equal to the threshold. The MAD-based SLO distance also exhibits the property of not being affected by a few large batch distance values (outliers). In contrast, the SD-based SLO distance is more sensitive to such values, given that it is based on squaring distances from the target.

4 Evaluation

4.1 Military Server Benchmark

The Military Server Benchmark (MSB) is an application developed by Araujo et al. [2] designed to leverage accelerated computing in stream processing. The benchmark is composed of heavy computations and allows for the exploitation of data parallelism within each input item (i.e., inputs are records of attributes also called tuples) or in batches (depending on the size of the tuples used). The problem domain involves allocating military units on a map while considering the location requirements specific to each unit type. Drones fly over designated

coordinates of the map and collect data that will be used to allocate their assigned units. The system performs the required computations to allocate the military units efficiently using the data continuously received from the drones.

The application is structured as a pipeline of five computational stages. The first and the last are devoted to generating data and gathering results (I/O-bound source and sink stages). The three internal stages are each one parallel. The first executes a compute-intensive *map* pattern running on GPU, which extracts information for each coordinate explored by the drone. The second still runs on GPU and is based on the *map-reduce* pattern to find the most suitable coordinate for each military unit. The last stage is lightweight and still done by GPU. It performs final data validation.

The MSB implementation used in this work has been parallelized with FastFlow [1] and GSPARLIB [8]. FastFlow assembles and coordinates the pipeline stages, while GSPARLIB is responsible for the GPU offloading inside the stages. In the MSB pipeline, each tuple carries data from a specific drone. Batches are allocated in the GPU memory at the source stage and deallocated at the sink stage. As a result, batch sizes are defined for the entire pipeline rather than individually stage by stage.

Runtime performance variability in a SPA can be attributed to variables such as tuple inter-arrival time and tuple processing time. The time series generated by the moving average of these variables often demonstrates non-stationarities in real-world scenarios, such as increasing or decreasing trends and cyclic behaviors such as seasonal patterns. The overall software architecture of MSB is capable of reproducing workload variations called *computation patterns* applied to the computation time per tuple by the system (while the input rate is kept fixed for each execution). This is a realistic scenario for MSB, since drones generate data and transmit them at fixed rate. The algorithms to generate the computation patterns were based on [6]. Since they were originally designed for generating different frequency patterns of data arrivals, we adapted them to reproduce time-varying patterns affecting the tuple computation time.

We adopted a feedback control strategy [9] that turns the MSB pipeline into a closed loop in which the sink measures the end-to-end latencies of computed batches and transmits the measurements to the source stage. The source is responsible for receiving the measured latencies and applying the adaptation algorithm to decide the size for the next batches. The feedback loop should not result in additional delays for the source stage to process the incoming tuples. Hence, if the updated latencies did not arrive in time to decide the size of the next batch, the batch is delivered with the last computed size without blocking the data flow through the pipeline. We also perform sampling of the batch latencies, where the average of the samples is provided as input to the adaptation algorithm. Sampling is controlled by a parameter named *sample size*. This parameter can be used to try reducing interferences from specific batches containing latencies that significantly deviate from their neighbors.

Stein et al. [9] presented four adaptation algorithms for adapting the batch size, namely Fixed Adaptation Factor (FAF), Percentage-Based Adaptation Fac-

tor (PBAF), PBAF without threshold (PBAF-WT), and Multiplier-Based Adaptation Factor (MBAF). The algorithms accept as parameters a target latency (our SLO), a threshold region around the target latency, the current batch size, and step size (called adaptation factor by the authors). The goal is to regulate the batch size to keep the target latency inside the threshold region, which is expressed in percentage.

Regarding the main differences among the algorithms, the most straightforward is FAF. FAF increments or decrements the batch size by a fixed step when the actual latency exceeds the lower or upper bound respectively. PBAF behaves similarly but reduces the (user-specified) step size as it approaches the target latency, aiming for more precision and likely avoiding stepping out of threshold bounds. PBAF-WT focuses solely on the target, resulting in frequent batch size changes to improve precision but risking going out of bounds with larger step sizes. MBAF prioritizes reaching the threshold region quickly by scaling the step size based on the distance from the target, aiming at offering a faster response to workload variations.

4.2 Evaluation Results

In this section we present the results of our evaluation of the algorithms proposed by [9] applied in the Military Server Benchmark [2]. The experiments were executed in a computer equipped with an AMD Ryzen 5 processor (6 cores and 12 threads) and 32 GB of RAM. The GPU was an NVIDIA GeForce RTX 3090 (Ampere architecture) with 24 GB of VRAM and 10,496 CUDA cores. The operating system was Ubuntu 20.04 LTS. The software used was GCC 9.0.5, CUDA 11, FastFlow 3, and an optimized version of GSPARLIB provided by [2]. We used the GCC compiler-level optimization 3 (flag `O3`). There was no replicated stage in our stream processing pipeline, so each stage is run by a dedicated host thread offloading computation on GPU.

We execute MSB ten times for each parameter combination to obtain the means and standard deviations. The evaluation parameters are as follows:

- Target latency (SLO): 3 milliseconds
- Threshold value: 5%
- Step sizes (adaptation factors): 1, 5, 10, 15, and 20
- Latency sample sizes: 1, 5, 10 and 20
- Adaptation algorithms: FAF, PBAF, PBAF-WT, and MBAF

The execution time for processing our workload (containing 500k tuples) varied from 6.4 seconds to around 1 minute and a half, depending on the parameters used. We consider the whole execution in the experiments, without warm-up or cool-down periods. We select the parameter values based on the characteristics of the MSB in the chosen hardware platform. The three-millisecond target latency has been chosen because empirically it is achievable at practically any point in the execution if the right batch size is applied by the system. It is (approximately) the highest tuple latency encountered when executing MSB without

batching, such that it can be achieved with a batch size close to one in the most compute-intensive regions, while the lowest tuple latencies require batch sizes of around 300 tuples. The threshold values are kept the same as in [9].

Figure 1 presents the best results for the metrics discussed in Sect. 3. For each algorithm, the metrics values (in percentage) from the best batched and itemized SLO hits are shown, as well as the best MAD-based and SD-based distances. They are labelled using the abbreviations for the definitions in Sect. 3. Furthermore, in Tables 1 and 2 we present the metrics values, as well as the configuration (step size and latency sample size) used for achieving those metrics.

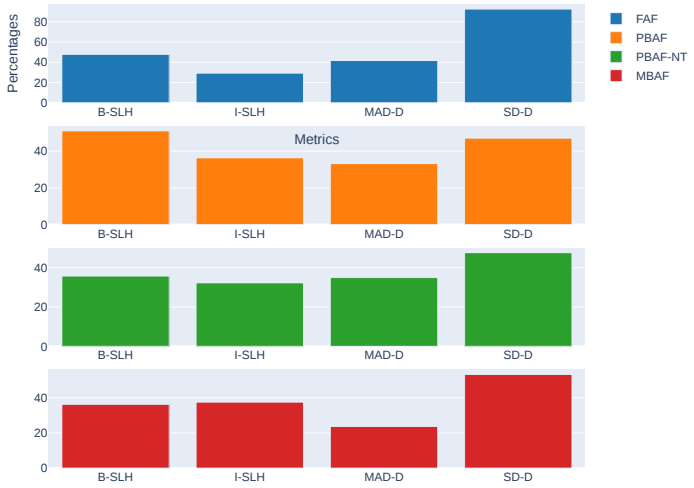

Fig. 1. Best SLO hit and distance metrics per algorithm.

Table 1 presents the best SLO hit metrics achieved by each algorithm, as well as the configuration used to achieve the metric value.

Table 1. Best SLO Hit metrics and configurations by algorithm.

Algo.	Batched SLO Hit			Itemized SLO Hit		
	VALUE	STEP	SAMPLE	VALUE	STEP	SAMPLE
FAF	47.83	1	1	29.25	1	1
PBAF	51.02	5	1	36.35	10	1
PBAF-WT	35.72	5	5	32.28	5	1
MBAF	36.30	5	5	37.55	1	1

The algorithms were able to meet the latency SLO for 29% (FAF) to 37% (MBAF) of the 500K processed tuples. The SLO hit per batch ranged between 32% and 51%, with MBAF and PBAF-WT defining the lower end, and FAF and PBAF defining the higher end of the range. The best batched SLO hit results

achieved in [9] for the 5% threshold range from 50.58% to 90.39%. The highest batched SLO hit we achieve in MSB with the same threshold is 53.452%, which is comparable with the lowest result from [9]. These results demonstrate that our workload is more challenging than the previous one from [9].

The algorithms achieve comparable results across all SLO hit metrics. However, the configuration they use to achieve their best results is not homogeneous. A step size of 5 or 10 yielded the best SLO hit results for PBAF and PBAF-WT, owing to their ability to use a fraction of the step size as they approach the threshold bounds. Conversely, the best SLO hit results for FAF and MBAF are achieved with a step size of 1 in most cases. This happens because larger step sizes for these algorithms, while allowing a faster reaction, also prevent them from fine-tuning the batch size when latencies are close to the target.

In Table 1 a consistent pattern can be discerned where the latency sample sizes are always 1 for the itemized SLO hit, while for the batched SLO hit they are set to 5 in half of the instances (rows). This behavior stems from a trade-off between two conflicting situations that favor one metric over the other. The first situation is that a latency sampling greater than 1 increases the SLO hit in regions with very frequent latency variations, such as in the workload segments belonging to the (gradually) increasing and decreasing computation patterns depicted in Figure 2 across the green line. However, (as in the second situation) this increased sampling reduces reactivity since five latency samples must be collected before re-evaluating the strategy again. Consequently, the SLO hit will be lower in regions with abrupt latency changes necessitating high reactivity, e.g., spike and binary patterns. Furthermore, these patterns contain extensive regions with minimal computation, where large batch sizes (close to 300) are needed to keep the SLO. Specifically for the itemized SLO hit, batches missing the threshold bounds in these regions result in a greater cost than for the batched SLO hit. Consequently, the best results for the itemized SLO hit do not incorporate latency sample sizes greater than 1. Table 2 presents the best distance metrics achieved by each algorithm, as well as the configuration used to achieve the metric value.

Table 2. Best distance metrics and configurations by algorithm.

Algo.	MAD-based Distance			SD-based Distance		
	VALUE	STEP	SAMPLE	VALUE	STEP	SAMPLE
FAF	41.78	5	1	92.64	20	1
PBAF	33.2	10	1	47.04	1	20
PBAF-WT	34.95	1	5	47.61	1	5
MBAF	23.73	5	5	53.3	1	20

Regarding the distance metrics, the best results for the MAD-based distance are similar for all algorithms except MBAF, predominantly staying within the range of the first five digits of 30. MBAF presented MAD-based distances around 25% smaller than the other algorithms. We attribute this behavior mainly to its high reactivity to spikes in latency.

The best configurations in terms of distance metrics do not necessarily match the best configurations in terms of SLO hit metrics. In Table 1, the maximum latency sample size used was 5. However, Table 2 includes configurations under the SD-based distance metric where the combination of the smallest step size (1) and the largest sample size (20) leads to significant delays in adaptation combined with minimal adjustments in batch size. This results in executions where the batch latencies converge towards the target from the lower threshold bound, albeit failing to cross into the threshold bounds most of the time. Figure 2 illustrates this behavior across the red lines. Keeping the latency closer to the lower threshold bound requires smaller batch sizes, which consequently contributes to avoiding extreme latency spikes caused by large-sized, computation-heavy batches. Another successful strategy for mitigating spikes is the use of large step sizes combined with small sample sizes to increase reactivity. Although the latency spikes are high with this strategy, they are reduced more quickly to a latency closer to the target. We do not observe the largest tested sample sizes being used in Table 2 for the MAD-based distance because this metric does not penalize latency spikes as much as the SD-based distance does. However, we observed that smaller values of the former are more indicative of greater SLO hits, an observation that does not always hold for the latter.

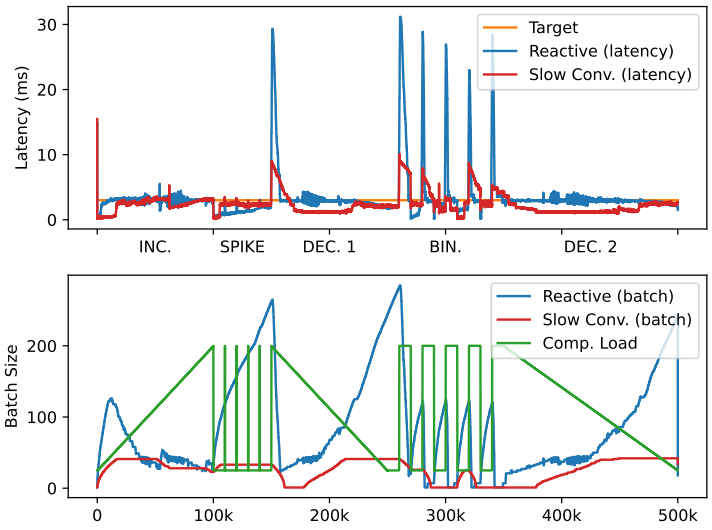

Fig. 2. Latency, batch size and computational load per item.

Figure 2 depicts the two most successful strategies concerning the SLO hit and distance metrics. We use strategy as a general term referring to the combinations of algorithms and configurations that generate a specific adaptation behavior. The strategies we chose to show are particularly remarkable due to their

effectiveness in mitigating latency spikes through different means. The strategy we named *Reactive* comes from MBAF, which attained the best itemized and batched SLO hit, and the best MAD-based distance. The metrics and configuration for this strategy can be found in Table 1 in the row corresponding to MBAF. It is the same configuration found in Table 2 under the MAD-based distance. The second strategy, which we named *Slow Conv.*, comes from PBAF. It can be found in Table 2 under SD-based distance, having achieved the best result in this metric.

Figure 2 depicts the latency and the choice of batch sizes for two different algorithms and configurations along an execution time representation. In the x-axis, *time* is normalized across all executions, progressing based on a numerical identifier (id) associated with each tuple, according to the order that it was produced (which is fixed). Regarding the y-axis, the red lines represent the reactive configuration, labelled *Reactive*. The blue lines represent the slow convergence configuration, labelled *Slow Conv.*. The orange line represents the target latency.

The green line (named *Comp. Load*) is meant to represent the computational load of the item. It provides a (theoretical/conceptual) notion of the computation patterns (Sect. 4.1). For instance, if there is an increasing load, we will usually observe the batch size lines decreasing in an attempt to offset it. We name each pattern in Figure 2. The abbreviated names are *Inc* for increasing, *Dec* for decreasing, and *Bin* for binary.

Summary. Our key findings can be summarized as follows. Firstly, the algorithms mostly achieve comparable results (although using different configurations) for the SLO hit metrics, while MBAF stands out mainly in the MAD-based distance. Secondly, the itemized SLO hits were consistently lower than the batched SLO hits, indicating that relying solely on the latter metric to assess SLO compliance can be misleading. Thirdly, the SLO hit, and distance metrics demonstrate that more reactive algorithms (such as MBAF) and configurations produce superior results. Lastly, the distance metrics indicate that less responsive configurations attaining low SLO hits can still be close to the threshold and mitigate large latency spikes.

5 Conclusion

In this paper we presented an evaluation of four self-adaptive algorithms for stream processing with GPUs from Stein et al. [9]. We implemented the self-adaptive strategy and algorithms with FastFlow [1] and GSPARLIB [8] in the Military Server Benchmark application [2].

One of the current limitations of the proposed algorithms is the exclusive focus on latency. Having only the latency as a target creates situations in which the batch size is increased with the sole purpose of artificially increasing latency, e.g., there is no gain in terms of throughput. This is not desirable, given that the reason for increasing latency is to achieve a greater throughput. In such cases, it would be preferable to disregard the lower threshold bound. Another

notable limitation arises from the tradeoff presented by the current algorithms between reactivity (MBAF) and finer tuning near threshold bounds (PBAF and PBAF-WT). In future work, we plan to explore new approaches to overcome the aforementioned limitations of the evaluated algorithms, by targeting other SLOs besides latency, and by exploring adaptation algorithms that can adequately achieve both fine-tuning and reactivity.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280. John Wiley & Sons, Ltd (2017)
2. Araujo, G.A.d., et al.: Data and stream parallelism optimizations on GPUs. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul (2022)
3. Cheng, D., Zhou, X., Wang, Y., Jiang, C.: Adaptive scheduling parallel jobs with dynamic batching in spark streaming. IEEE Transactions on Parallel and Distributed Systems **29**(12), 2672–2685 (2018)
4. Das, T., Zhong, Y., Stoica, I., Shenker, S.: Adaptive stream processing using dynamic batch sizing. In: Proceedings of the ACM Symposium on Cloud Computing. p. 1–13. SOCC '14, Association for Computing Machinery, New York, NY, USA (2014)
5. 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 **7**, 48753–48769 (2019)
6. Garcia, A.M., Griebler, D., Schepke, C., Fernandes, L.G.L.: Evaluating micro-batch and data frequency for stream processing applications on multi-cores. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 10–17. IEEE (2022)
7. Rockenbach, D.A., Stein, C.M., Griebler, D., Mencagli, G., Torquati, M., Danelutto, M., Fernandes, L.G.: Stream processing on multi-cores with gpus: Parallel programming models' challenges. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 834–841 (2019)
8. Rockenbach, D.A.: High-level programming abstractions for stream parallelism on gpus. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul (2020)
9. Stein, C.M., Rockenbach, D.A., Griebler, D., Torquati, M., Mencagli, G., Danelutto, M., Fernandes, L.G.: Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. vol. 33, p. 5786 (2021)
10. Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M.J., Recht, B., Stoica, I.: Drizzle: Fast and adaptable stream processing at scale. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 374–389. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017)
11. Vogel, A., Griebler, D., Danelutto, M., Fernandes, L.G.: Self-adaptation on parallel stream processing: A systematic review. Concurrency and Computation: Practice and Experience **34**(6), e6759 (2022)
12. Zhang, Q., Song, Y., Routray, R.R., Shi, W.: Adaptive block and batch sizing for batched stream processing system. In: 2016 IEEE International Conference on Autonomic Computing (ICAC). pp. 35–44 (2016)