High-Throughput Asynchronous Convolutions for High-Resolution Event-Cameras

Conference Paper · June 2022

DOI: 10.1109/EBCCSP56922.2022.9845500

5 authors, including:

Leandro De souza Rosa
Delft University of Technology
14 PUBLICATIONS **30** CITATIONS
SEE PROFILE

Chiara Bartolozzi
Istituto Italiano di Tecnologia
106 PUBLICATIONS **3,800** CITATIONS
SEE PROFILE

Arren Glover
Istituto Italiano di Tecnologia
34 PUBLICATIONS **831** CITATIONS
SEE PROFILE

High-Throughput Asynchronous Convolutions for High-Resolution Event-Cameras

Leandro de Souza Rosa
Event-Driven Perception for Robotics
Italian Institute of Technology
Genova, Italy
leandro.desouzarosa@iit.it

Aiko Dinale
Event-Driven Perception for Robotics
Italian Institute of Technology
Genova, Italy
aiko.dinale@iit.it

Simeon Bamford
Event-Driven Perception for Robotics
Italian Institute of Technology
Genova, Italy
simeon.bamford@iit.it

Chiara Bartolozzi
Event-Driven Perception for Robotics
Italian Institute of Technology
Genova, Italy
chiara.bartolozzi@iit.it

Arren Glover
Event-Driven Perception for Robotics
Italian Institute of Technology
Genova, Italy
arren.glover@iit.it

***Abstract*—Event cameras are promising sensors for on-line and real-time vision tasks due to their high temporal resolution, low latency, and redundant static data elimination. Many vision algorithms use some form of spatial convolution (i.e. spatial pattern detection) as a fundamental component. However, additional consideration must be taken for event cameras, as the visual signal is asynchronous and sparse. While elegant methods have been proposed for event-based convolutions, they are unsuitable for real scenarios due to their inefficient processing pipeline and subsequent low event-throughput. This paper presents an efficient implementation based on decoupling the event-based computations from the computationally heavy convolutions, increasing the maximum event processing rate by $15.92\times$ to over $10\ million$ events/second, while still maintaining the event-based paradigm of asynchronous input and output. Results on public datasets with modern 640×480 event-camera recordings show that the proposed implementation achieves real-time processing with minimal impact on the convolution result, while the prior state-of-the-art results in a latency of over 1 second.**

***Index Terms*—Event-Driven Cameras, Asynchronous Processing, Spatial Convolutions, Real-Time Processing**

I. Introduction

Spatial convolutions are arguably among the most common computations in computer vision, used in feature detection and learning pipelines. The convolution operator takes an image frame and a kernel as inputs, and outputs a convolved image that is typically computed pixel-by-pixel by multiplying and accumulating image regions and the kernel.

In recent years, event cameras have become popular among the robotics community. In contrast to traditional cameras, which output synchronous image frames, event-cameras output an asynchronous stream of events, encoding relative changes in the brightness of individual pixels. This encoding supports extreme low-latency, high temporal precision, and dynamic range while drastically reducing redundant data that needs to be transmitted and processed [1]. As such, they are a key enabling technology for real-time applications in highly dynamic scenarios [2].

Traditional computer vision approaches to convolutions cannot be directly applied over the output of such sensors; convolutions require an image, while the events are sparse and asynchronous. Solutions typically fall either into synchronous or asynchronous approaches.

An example of a synchronous approach is to create a virtual frame by accumulating incoming events while "leaking" information from past events [3]. The virtual frame can be used by traditional object recognition pipelines that output the result of convolution synchronously. While the input can be asynchronous (event-by-event) or synchronous (creating images from batches), the important aspect is that the *output* is synchronous. I.e., the object recognition can only occur at the fixed rate of the frame reconstruction, and therefore all downstream processing becomes synchronous as well.

An example of an asynchronous approaches is a convolutional neural network that accepts asynchronous events as input and updates the neuron activation values sparsely [4]. Such asynchronous approaches perform algorithmic computation for each event, e.g., propagating signal through multiple convolution layers in a network. On traditional CPU hardware, increasing the amount of computation per event reduces the number of events processed per second, that is the *event-throughput*. A single-layer, asynchronous convolution was proposed by [5] which we show to process approximately $1\ million$ Events per Second (e/s). However, even VGA event cameras can easily surpass this event rate, and newer cameras produce over $10\ million$ e/s. If the event-rate surpasses the algorithm's *event-throughput*, the algorithm does not run in real-time.

Leandro de Souza Rosa is currently affiliated with the Cognitive Robotics Group, Delft University of Technology, Delft, The Netherlands <l.desouzarosa@tudelft.nl>

978-1-6654-5349-3/22/$31.00 ©2022 IEEE

This work proposes a method to address the problem of how a single layer of convolution can be performed asynchronously with a high *event-throughput*, being suitable for VGA or even higher resolution cameras. It enables asynchronous convolution-based algorithms, such as corner detection proposed in [5], to run on-line and in real-time. Also, it has possible implications for real-time multi-layer convolution pipelines (e.g., spiking neural networks) as the asynchronous output allows asynchronous events propagation beyond the first layer.

To achieve both high event-throughput and asynchronous convolution, we propose an unconventional hybrid of the synchronous and asynchronous methods. To do so, we follow previous work, which operates in the same way for the single algorithm of Harris corner detection [6]. However, we show that the principle applies to any convolution and, therefore, has more widespread applicability.

The key aspect of the asynchronous event-by-event pipeline is that an output event is required to be created for each incoming event, containing the convolution result. However, it is infeasible to compute convolutions for each, even considering sparse ones. The core novelty is that we instead assign the convolution result from a convolved image from *sometime in the past* with a correction factor applied. The convolved image is updated As Soon As Possible (ASAP) in a separate thread outside of the asynchronous processing pipeline, and therefore the convolution operation does not impact the *event-throughput*. Therefore, the fast asynchronous process of computing the correction factor and the slower convolution update are *decoupled*, so that the part that depends on the number of events has small impact of the computational time. The speed-up of the computation has a small cost in terms of precision, as the convolution result applied to the event is an approximation as it is taken from a not-up-to-date* convolved image.

In this paper, we describe how to switch from a reference asynchronous event-by-event convolution [5] to the proposed hybrid method, that speeds up the implementation while maintaining asynchronous inputs and outputs. We report *event-throughput*, latency and accuracy, when running both implementations on publicly available high-resolution event-camera datasets.

II. Reference Implementation

We constrain our in-detail description of relevant algorithms in the literature to the event-by-event convolution algorithm proposed in [5] as it is at the core of the problem addressed by this paper and will be referred to as the *reference* algorithm, against which to compare our implementation. The equations are stated in the Appendix.

An event camera has asynchronous pixels, which independently react to changes in the field-of-view brightness [7]. The (x, y) pixel fires an event when its log intensity (I) changes above a threshold (s) relative to reference value (I_{ref}), as described in Equation 1. After firing, $I_{ref}(x, y)$ is set to $I(x, y)$.

*on the order of 1 ms

$$|\log(I(x,y)) - \log(I_{ref}(x,y))| > s \tag{1}$$

An event is defined as $e_i = \{x_i, y_i, t_i, p_i\}$, where t_i is its timestamp, and p_i its polarity. Each asynchronous input event produces an output event, $\tilde{e}_i = \{x_i, y_i, t_i, c_i\}$, where c_i is the intensity image convolved with desired convolution kernel W at that pixel position. We write $\mathcal{C} = W \circledast I$ for the convolved intensity image, and $c_i = \mathcal{C}(x_i, y_i)$.

In the reference implementation, each event requires an internal representation $\mathcal{C}^\star$ to be updated for a $kernelSize$ patch around the event location (x_i, y_i). This patch-wise update leads to regions updated at different points in time, which is denoted by the "$\star$" symbol, and will be referred to as "out-of-sync".

The update has three components, an integration of event polarity, a temporal decay, and the convolution with the kernel W, as defined in Algorithm 1. The temporal decay is influenced by a user-defined scene-dependent parameter α, and the elapsed time between the current and the previous updates of $\mathcal{C}^\star$, computed with an auxiliary matrix T, which saves the "last update time" for each pixel.

The output of Algorithm 1 forms an asynchronous signal, similar to the input event-stream, and can be used for event-based computations. E.g., the Harris corner detection [5], corner tracking methods [8], or 3D reconstruction methods [9].

Algorithm 1: The (u, v) index indicate a position in the patch of the respective matrix corresponding to the relative position in W. The output is an asynchronous, sparse, event-based convolution.

for $(u, v) \in patch(x_i, y_i)$ **do**
  $dt \leftarrow t_i - T(u, v)$;
  $\mathcal{C}^\star(u, v) \leftarrow \mathcal{C}^\star(u, v)e^{-\alpha dt} + W(u, v)p_i s$;
  $T(u, v) \leftarrow t_i$;
$\tilde{e}_i \leftarrow \{x_i, y_i, t_i, \mathcal{C}^\star(x_i, y_i)\}$;

Fig. 1 illustrates the asynchronous pipeline performing all updates of $\mathcal{C}^\star$ and producing the precise result $\tilde{e}_i$.

III. The Proposed High-Throughput Implementation

The proposed implementation again produces an asynchronous output $\tilde{e}_i = \{x_i, y_i, t_i, c_i\}$ for each input event from the camera but decouples two of the main operations to reduce the amount of per-event computation as much as possible. The key idea behind the asynchronous pipeline is that each output event is assigned the convolution score from a convolution image produced by a secondary process. The secondary process does not interrupt the processing flow of the asynchronous pipeline, and therefore the asynchronous *event-throughput* is maximised. However, the convolved image production is temporally delayed from the precise timing of the events, and therefore the assigned c_i value is an approximation. To offset the approximation error, a correction factor, ν, is added to the convolution score of each event.

The asynchronous pipeline operates similarly to the reference algorithm and is described in Algorithm 2. However, as

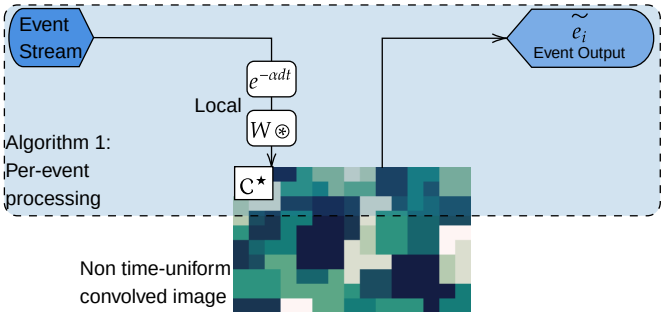


Fig. 1: Overview of the reference implementation. $kernelSize$ patches of the convolved image $\mathcal{C}^\star$ are updated for each event resulting in an out-of-sync image represented by the different coloured patches (a darker-coloured pixel represents a more recent update).

the reference algorithm directly updates $\mathcal{C}^\star$ for each event, Algorithm 2 updates only the polarity integration and decay image for a single pixel, making the internal state represents the "out-of-sync" intensity image $I^\star$. The processing per event is reduced as only a single-pixel needs to be updated, and the convolution operation is not applied per event.

The secondary process performs the full image synchronisation and convolution, creating a time-synchronised image $\mathcal{C}$ from the most up-to-date $I^\star$. The steps described in Algorithm 3 indicate the temporal decay for all pixels in the image and the convolution. Algorithm 3 is performed ASAP to obtain the most up-to-date convolution result, which is sampled to assign the convolution output score in Algorithm 2.

Algorithm 2 and Algorithm 3 are processed in separate threads, using different cores of a CPU. Both algorithms run as fast as possible, and there is no blocking operation between them. Algorithm 2 does not wait for Algorithm 3 to complete in order to assign scores to the output events. The convolution result must be taken from $\mathcal{C}$, which stores the correct convolution result, *but for some point in the past.* The error estimate (ν_i) comes from this temporal discrepancy between the exact time and the last point in time when $\mathcal{C}$ was calculated.

For any event, the estimate can be improved given the knowledge that, at minimum, the single incoming event has not been included by the convolution operation. Therefore, simply adding the $\nu_i = sp_iW_o$ to the convolution result $C(x_i, y_i)$ gives an improved convolution estimate and helps to offset any errors, where W_o is the value of the kernel at its centre.

The correction factor cannot account for the contribution of other events within the spatial region of the $kernelSize$ since the last update of $\mathcal{C}$. Therefore, the error due to estimation would be more noticeable in highly dynamic scenes in which multiple events occur in the same region in a very short period while it will be smaller in slow dynamic scenes.

The derivation of these algorithms from the reference algorithm is described in the Appendix.

Algorithm 2: The proposed asynchronous event-by-event update, which updates a single pixel in $I^\star$, and assigns the output convolution value from $\mathcal{C}^\star$, which is computed in Algorithm 3.

$dt \leftarrow t_i - T(x_i, y_i)$;
$I^\star(x_i, y_i) \leftarrow I^\star(x_i, y_i)e^{-\alpha dt} + p_i s$;
$T(x_i, y_i) \leftarrow t_i$;
$\tilde{e}_i \leftarrow \{x_i, y_i, t_i, \mathcal{C}(x_i, y_i) + \nu_i\}$;

Algorithm 3: The proposed decoupled secondary process to calculate $\mathcal{C}$ outside the event-by-event pipeline.

$I \leftarrow e^{-\alpha(t_\omega - T)} I^\star$;
$\mathcal{C} \leftarrow W \circledast I$;

IV. Experiments and Results

The scope of the experiments is to compare the proposed high-throughput implementation of asynchronous convolutions with the reference implementation. We present: (i) results for the algorithm *event-throughput*; (ii) the subsequent latency when processing publicly available datasets, which are typical conditions in which an event-camera could be used; (iii) an evaluation of the error introduced by the approximation required by the proposed algorithm; (iv) an evaluation of the latency when the full image frame is required as an output (instead of asynchronous events).

We avoid presenting results for different kernels and visualising different outputs. As described in the Appendix, the equation to produce a time-synchronised image $\mathcal{C}$ for both methods is mathematically identical. Therefore, we can focus on the *event-throughput* and the approximation error.

A. Benchmarks and Experimental Setup

To test the approaches, we used the "High Speed and HDR Datasets" recordings [10][†] from a Samsung Dynamic Vision Sensor (DVS) Gen3 event camera (640×480 pixels resolution and maximum throughput of 300×10^6 e/s [11]), which are presented in Table I.

TABLE I: Sequences names and abbreviations for the "High Speed and HDR Datasets" recordings [10], containing recorded sequences with high temporal resolution, complex texture, and high luminosity contrast and variation.

Sequence	Abbreviation
Gun Bullet Gnome	gn
Gun Bullet Mug	mu
HDR Selfie	se
HDR Sun	su
HDR Tunnel	tu
Popping Air Balloon	ai
Popping Water Balloon	wa

[†]Asset under GNU General Public License v3.0.

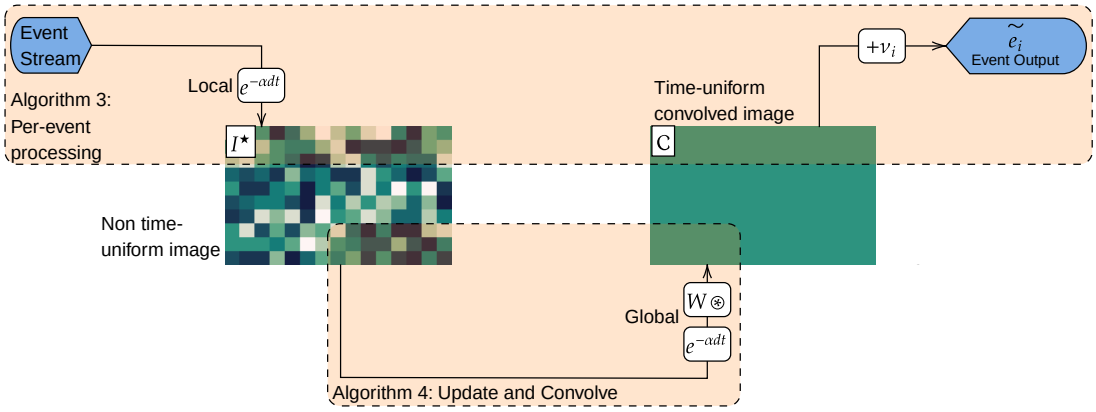


Fig. 2: Overview of the proposed implementation. The image $I^\star$ is updated per event. Each pixel represents a different point in time (a darker-coloured pixel represents a more recent update). While $\mathcal{C}$ is computed ASAP, more events can arrive and update $I^\star$, meaning that $\mathcal{C}$ always has a delay in comparison to the newest pixel in $I^\star$.

The communication between the event-stream and convolution computations is implemented using YARP [12], while the computations are performed and measured in a single C++ executable. YARP and the network are used to stream the events in real-time as if produced live from a camera, eliminating delay inconsistencies with fully offline processing. YARP does not influence the computation and throughput result. All tests were run on an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz.

We chose a single Gaussian kernel for evaluation with parameters of $kernelSize = 3$, $\sigma = 0.3$, and $\alpha = \pi$, $s = 1$, and $p_i \in \{-1, 1\}$. Note that the computation time does not depend on the kernel type (e.g., Gaussian, Sobel, Box),

B. Event Processing Rate

The *event-throughput* for the reference and proposed implementations were measured as 0.61×10^6 e/s and 9.77×10^6 e/s, respectively, averaged over all data for all datasets, as shown in Figure 3. The $15.92\times$ average increases the threshold for which real-time operation can occur. The result is significant as VGA event-cameras produce 10 $million$ e/s, depending on the speed of motion. Therefore, the proposed algorithm can directly enable real-time operation in typical conditions, which would not be possible with the prior state-of-the-art reference implementation.

C. Latency

The sequences are played back with precise event-timing, as if produced by a live camera using YARP, allowing to evaluate the latency under typical conditions. In this way, the algorithm cannot process events ahead their time, and delays will occur if the event rate of the dataset is larger than the algorithm's *event-throughput*, i.e., the algorithm becomes not real-time.

The benchmark sequences produce events with up to 35×10^6 e/s, and Figure 4 shows that the proposed implementation maintains a latency of the order of milliseconds on all sequences. The spiking pattern in the delay for our method is caused by the discretisation of packet timestamps by YARP, whose communication occurs with packets of multiple events;

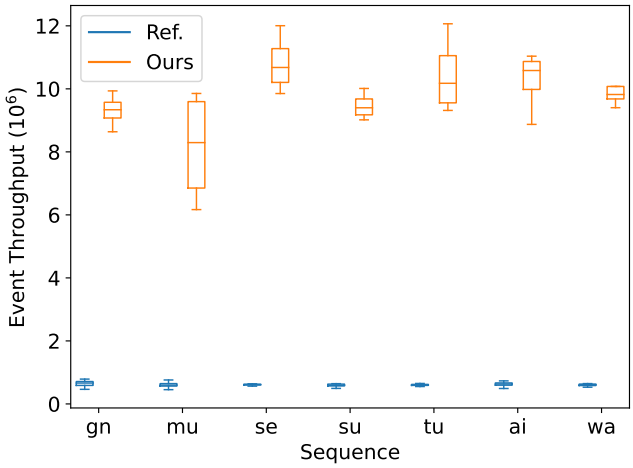
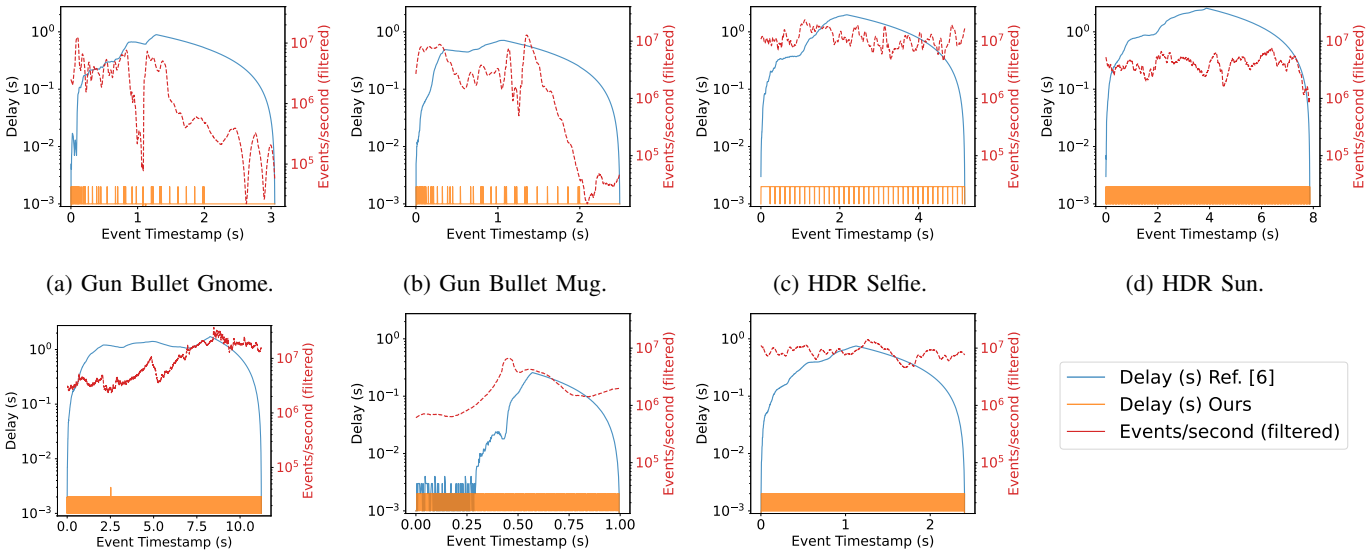


Fig. 3: Event throughput ($\times 10^6$ e/s) for the proposed and reference implementations.

however, real-time execution still occurs as the delay does not grow over time.

In contrast, the reference algorithm produces delays in the order of several seconds, demonstrating it is infeasible for real-time operation with the typical scenarios found in the datasets.

For each sequence in Figure 4, there is a point at which the delay of the reference algorithm begins to decrease. At this point, the dataset has "finished", but the algorithm is still processing previous data to "catch-up" to real-time. For example, in Figure 4d the delay starts dropping at $timestamp \approx 4$ s meaning that at ≈ 8 s of wall-clock time (end of the real-time event-stream) the reference implementation was still processing the event generated at $timestamp \approx 4$ s. Particularly, Figure 4f shows that the reference algorithm starts to accumulate delays (shortly before $timestamp = 0.3$ s) when the event rate reaches 10^6 e/s, implying that this is the maximum event rate



(a) Gun Bullet Gnome. (b) Gun Bullet Mug. (c) HDR Selfie. (d) HDR Sun.

(e) HDR Tunnel. (f) Popping Air Balloon. (g) Popping Water Balloon.

Fig. 4: Delay for reference [5] (blue) and proposed (orange) methods and camera throughput (red) according to the timestamp of the events. The delay of the reference implementation grows until all events are received, falling from that point onward. Note the log scale.

that the reference implementation can cope and still maintain real-time execution (in our experimental setup). Please refer to our video in the supplementary material for better visualisation of the latency.

D. Approximation Error

The proposed approach achieves a higher event throughput, with a trade-off on the accuracy of the result. This error can be quantified per event at an asynchronous time-scale, comparing the convolution output assigned to each event between the reference and proposed algorithms. We, therefore, compare $\tilde{e}_i$ values obtained with Algorithm 1 and Algorithm 2.

The average error from the proposed to the reference implementation was $< 0.5\%$. Figure 5 shows the distribution of the absolute error w.r.t. the reference implementation, given by $error_i = |\tilde{e}_i^{ref} - \tilde{e}_i^{our}|$, and also the reference convolved image intensity. The error is non-zero as typically more than one event occurs in any given $kernelSize$ patch between updates of $\mathcal{C}$. Therefore, the correction factor ν_i is not able to account for all sources of error. However, the error is still minimised, and we propose that the 0.5% error is a worthwhile trade-off for the increase in the *event-throughput*.

To attempt to understand how the individual event error would affect any downstream application using the convolution result, we compute (i) the `ssim` metric [13], which compares luminosity, contrast and structure for assessing quality structural information degradation; and (ii) the error's impact in the *energy* of the $kernelSize$ patch around each event's position,

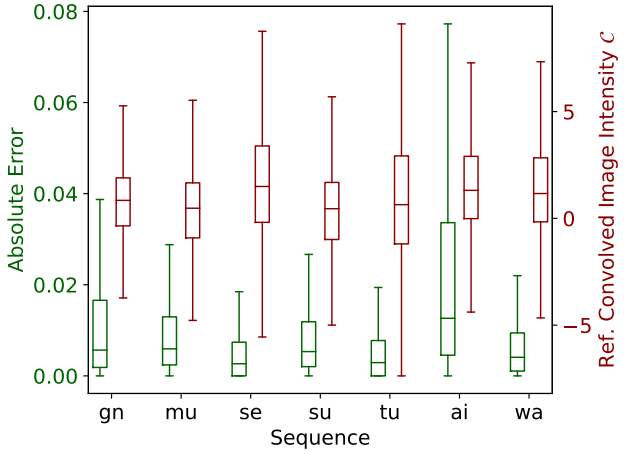


Fig. 5: Mean reference convolved image intensity $\mathcal{C}^{\star}_{ref}(x_i, y_i)$ at each event position, and absolute error $error_i$ between the reference and proposed output event. The error is typically two orders of magnitude smaller than the intensity values.

defined as Equation 2.

$$\epsilon_i = \frac{error_i}{energy} = \frac{error_i}{\sum_{j,k \in patch} |\mathcal{C}^{ref}(j,k)|} \tag{2}$$

Table II presents the error in the patch's energy and the `ssim` metric. Both metrics are measured on the local patch since the event output is only affected by events within its neighbourhood. In our tests, the error accounts for up to 1.11%

TABLE II: Geometric mean of the per-event impact in the patch's energy caused by the error between the proposed and the reference implementation, and average $ssim$ metric [13] over the patches.

Seq.	gn	mu	se	su	tu	ai	wa
ϵ (%)	1.11	0.6	0.02	0.15	0.06	0.91	0.1
ssim	0.70	0.73	0.89	0.76	0.88	0.80	0.89

TABLE III: Total time for reproducing the complete sequence frames and benchmark duration in seconds.

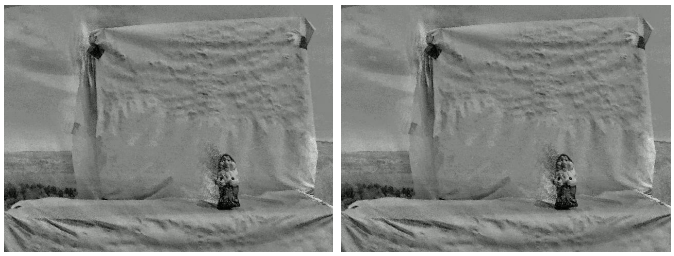
Seq.	gn	mu	se	su	tu	ai	wa
duration	3.06	2.49	5.17	7.86	11.23	1.00	2.41
ref.	636.98	468.07	1615.37	3545.82	4810.45	30.75	373.11
ours	3.16	2.52	5.24	7.90	11.32	1.07	2.48

of the patch's energy; the achieved range of values is shown to have minimal impacts on the robustness of image descriptors for texture classification [14].

E. Frame Rates and Inherited Delays

In addition to a purely asynchronous output, some applications may also have use for high-rate outputs of the synchronised, fully convolved frame, $\mathcal{C}$, which is visualised in Figure 6. The proposed approach calculates $\mathcal{C}$ as a by-product of the algorithm. However, [5] does not specify a policy for applying the whole image update, which requires synchronising all pixels in the image to an identical timestamp, i.e. $\mathcal{C}^\star$ must be converted to $\mathcal{C}$. For a fair comparison, we perform this operation in a second thread, similarly to our proposed approach, so it does not affect the asynchronous event-by-event pipeline.

We compute the time taken to reproduce the complete sequence of frames putting together the asynchronous and synchronous threads. Table III shows that the reference implementation takes $30\ s$ to reproduce $1\ s$ worth of events. Even though the reference implementation can produce frames faster ($\approx 450\ fps$) than the proposed implementation ($\approx 300\ fps$), as the reference algorithm has already performed the convolution step in Algorithm 1, the frames will inherit the latency from the event pipeline and be (for tested datasets) several seconds in delay.



(a) Ref. (b) Our.

Fig. 6: Example frames synchronized by the events from **gn** benchmark for a qualitative evaluation of the error.

V. Discussion

The experiments were performed with $kernelSize = 3$, giving the most favourable results to the reference algorithm as increasing $kernelSize$ leads to a $kernelSize^2$ drop in its *event-throughput*. For the proposed algorithm, the increase in $kernelSize$ does not change the *event-throughput*, but can instead increase the time taken to compute $\mathcal{C}$, subsequently increasing the approximation error.

In periods in which the event rate is less than the reference threshold, the use of the proposed algorithm will perform additional redundant computations, as $\mathcal{C}$ is updated with very little change in events. However, a throttling of the update rate of Algorithm 3 could easily be introduced when the event rate is low, saving energy.

Neither algorithm can guarantee real-time operation for all conditions (all event rates); instead, the increased throughput simply increases the threshold for which real-time can be achieved. In this paper, we discuss real-time *in regards to the publicly available high-resolution datasets tested* and demonstrate that the reference algorithm is not suitable for real-time processing of such datasets.

The proposed implementation naturally requires at least two processing units to fully explore the parallelism of the proposed separation between event-stream and convolution processing, which is reasonable since most modern processors provide multiple cores.

The reference implementation may be more suitable if the algorithm is implemented on specific (event-driven) hardware, which typically allows fully parallel operation for each event, and therefore decoupling event-processing is irrelevant. For example, [15] presents an application specific chip for computing pseudo-convolutions which achieves up to $16,7\ million$ e/s, while our implementation achieves up to $9,77\ million$ e/s for computing complete convolutions on a off-the-shelf computer. Nevertheless, such hardware is not readily available, and it is not simple to implement custom algorithms. Thus, the majority of event-based vision is still performed with CPU-based architectures. It is not easy to speculate on how a GPU architecture could speed up event-based computations, as they traditionally are designed to process synchronous spatial patterns while the events are serial and asynchronous.

The proposed implementation was modelled from the reference implementation in terms of the base method of handling event information, i.e. the integration of polarity images with exponential temporal decay. However, any method for producing the underlying image representation could be used, and the decoupling paradigm could be applied to achieve asynchronous output with convolutions.

More efficient convolution methods, such as the Fast Fourier Transform convolutions [16], can possibly profit from the proposed decoupling, being an interesting topic for future investigation.

Finally, an important use case of this algorithm could be Spiking Neural Networks (SNNs). To achieve a spiking output, a threshold should be applied to the convolution score instead

of outputting every event with a convolution score. Only events that surpass the threshold should be propagated to subsequent layers. If each layer is implemented as a decoupled process, real-time multi-layer SNN simulation could be achieved. Further investigation would need to be performed to understand if neural models, such as leaky-integrate-and-fire, could be modelled in addition to the convolutions.

VI. Conclusion

This paper presents an efficient spatial convolution pipeline for event cameras, which explores decoupling the event stream's processing from the computations required to compute the image convolutions. The proposed implementation maintains an asynchronous output, which incorporates a correction factor that reduces approximation errors to negligible amounts in most cases.

The implementation reduces the per-event computation requirements, achieving real-time performance on state-of-the-art, high-throughput, event-camera datasets. The prior state-of-the-art achieved less than 1×10^6 e/s, which is not sufficient for the majority of real applications, creating a bottleneck for any application that builds on top of convolution, such as feature detection or Convolution Neural Networks (CNNs). The result translates into several seconds of latency - for applications that want to use event cameras for their low latency, the algorithm is unsuitable.

We present a general framework for convolutions and demonstrate the results on a single kernel, to maintain the focus of the paper on the real-time implementation. However, we point the reader to [5] for a wide range of applications and also to [6] for a specific implementation of a similar framework for corner detection.

Supplementary Material

The video is available at https://doi.org/10.5281/zenodo.6476382. Source code and instructions for installing and running the experiments are available at https://github.com/event-driven-robotics/high-throughput-convolutions.

References

[1] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128x128 120 dB 15us latency asynchronous temporal contrast vision sensor," *IEEE journal of solid-state circuits*, vol. 43, no. 2, pp. 566–576, 2008. 1

[2] G. Gallego, T. Delbruck, G. M. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based Vision: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2020. 1

[3] M. Cannici, M. Ciccone, A. Romanoni, and M. Matteucci, "Asynchronous convolutional networks for object detection in neuromorphic cameras," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019. 1, 8

[4] N. Messikommer, D. Gehrig, A. Loquercio, and D. Scaramuzza, "Event-based asynchronous sparse convolutional networks," in *European Conference on Computer Vision*. Springer, 2020, pp. 415–431. 1

[5] C. Scheerlinck, N. Barnes, and R. Mahony, "Asynchronous Spatial Image Convolutions for Event Cameras," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 816–822, 2019. 1, 2, 5, 6, 7

[6] A. Glover, A. Dinale, L. De Souza Rosa, S. Bamford, and C. Bartolozzi, "luvHarris: A Practical Corner Detector for Event-cameras," pp. 1–1, 2021. 2, 7

[7] J. Huang, M. Guo, and S. Chen, "A dynamic vision sensor with direct logarithmic output and full-frame picture-on-demand," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4. 2

[8] J. Duo and L. Zhao, "An Asynchronous Real-Time Corner Extraction and Tracking Algorithm for Event Camera," *Sensors*, vol. 21, no. 4, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/4/1475 2

[9] H. Rebecq, G. Gallego, E. Mueggler, and D. Scaramuzza, "EMVS: Event-based multi-view stereo—3D reconstruction with an event camera in real-time," *Int. J. Comput. Vis.*, vol. 126, pp. 1394–1414, Dec. 2018. 2

[10] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza, "High Speed and High Dynamic Range Video with an Event Camera," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2019. 3

[11] H. E. Ryu, "Industrial DVS design; key features and applications," in *Conf. on Computer Vision and Pattern Recognition*, 2019. 3

[12] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006. [Online]. Available: https://doi.org/10.5772/5761 4

[13] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. 5, 6

[14] G. Kylberg and I.-M. Sintorn, "Evaluation of noise robustness for local binary pattern descriptors in texture classification," *EURASIP Journal on Image and Video Processing*, vol. 2013, no. 1, pp. 1–20, 2013. 6

[15] L. Camunas-Mesa, C. Zamarreno-Ramos, A. Linares-Barranco, A. J. Acosta-Jimenez, T. Serrano-Gotarredona, and B. Linares-Barranco, "An Event-Driven Multi-Kernel Convolution Processor Module for Event-Driven Vision Sensors," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 2, pp. 504–517, 2012. 6

[16] C. S. Burrus and T. Parks, "Convolution algorithms," *Citeseer: New York, NY, USA*, 1985. 6

Appendix

We derive the relationship between the reference and proposed implementations, presented in Algorithms 1, and 2 and 3, respectively. This formulation was first presented in [5], being rewritten in this section for completeness and describing the differences w.r.t. the proposed algorithm.

Let $E(t) = \{e_i \mid t_i \leq t\}$ be the set of all incoming events up to time t. Our goal is to express the intensity image as a time-continuous signal derived from the event stream.

Theoretically, each image pixel value is given by the summation over time of the polarity-intensity product ($p_i s$) of all events in that pixel. However, the initial conditions, the sensor noise, and integration drift are typically system unknowns, making a straightforward integration unfeasible for any typical use case. Formally, Equation 3 presents each pixel (x, y) of the image as a time continuous signal, being $\delta(t - t_i)$ the Dirac function and δ_{x,x_i} and δ_{y,y_i} Kronecker delta functions. The image convolved with the kernel W is expressed by Equation 4, where W is a 2D convolution kernel, and $\circledast$ indicates the convolution operator between two 2D matrices.

$$I(x, y, t) = \int_0^t \sum_{i \in E(t)} p_i s \delta(\tau - t_i) \delta_{x,x_i} \delta_{y,y_i} d\tau \tag{3}$$

$$\mathcal{C}(t) = W \circledast I(t) \tag{4}$$

Instead of computing Equations 3 and 4 for each event, the convolved image $\mathcal{C}(x, y, t)$ can be iteratively estimated through asynchronous sparse computations according to Equation 5 [5], where $\mathcal{C}$, $\mathcal{C}^\star$, T, and P_i are matrices of the same size as the image. We use (x, y) to denote the element of row x and

column y of the matrices. Here, T saves the timestamp of the last event with position (x_i, y_i), and $P_i = p_i s \delta(t - t_i) \delta_{x,x_i} \delta_{y,y_i}$. This sparse computation incorporates a high-pass filter with a cut-off frequency α, being an efficient iterative process since the updates are done in a $kernelSize$-patch around the incoming event, denoted by $patch(x_i, y_i)$.

$$\mathcal{C}^\star(u,v) = e^{-\alpha(t_i - T(u,v))}\mathcal{C}^\star(u,v) + [W \circledast P_i](u,v) \quad \forall (u,v) \in \ patch(x_i.y_i) \tag{5}$$

Even though Equation 5 encodes the whole image, the $W \circledast P_i$ operation is a local spatial convolution of the kernel with an image of zeros except at the event position. When an event arrives, only a patch with the same size as the convolution kernel ($kernelSize$) centred around the event position (x_i, y_i) needs to be updated, causing different image regions to represent the convolved image at different points in time. The "$\star$" symbol denotes this internal "out-of-sync" state.

In order to obtain the whole image frame at a consistent point in time, as in Equation 4, all pixels of $\mathcal{C}^\star$ must be updated. The update consists of applying the exponential decay in Equation 6 on the convolved image, which can be seen as a leaky integrator [3] implemented via a high-pass filter. The decay takes into consideration the time elapsed between the last update of each pixel of T and the desired time for the image t_ω (usually, the last event's timestamp), and the cut-off frequency of the filter α. On a practical side, α needs to be tuned to the scene, given that it balances the convolved image sharpness against how fast the pixels take to fade.

$$\mathcal{C}(x,y) = e^{-\alpha(t_\omega - T(x,y))}\mathcal{C}^\star(x,y) \quad \forall\ x,y \tag{6}$$

For our proposed implementation, it is convenient to rewrite Equations 5 and 6 moving the convolution operation outside of the per-event part. Moving $W\circledast$ out of Equation 5 leads to Equation 7, which is the per-event update without the convolution kernel and it is equivalent to an out-of-sync estimate of the intensity image $I^\star$. Moving $W\circledast$ into Equation 6 leads to Equation 8, which is the whole image frame update up-to time t_i and its convolution with W, and results in the estimate of the convolved image.

$$I^\star(x_i, y_i) = \left[e^{-\alpha(t_i - T(x_i,y_i))} I^\star(x_i, y_i) + p_i s\right] \tag{7}$$

$$\mathcal{C}(x,y) = W \circledast e^{-\alpha(t_\omega - T(x,y))} I^\star(x,y) \quad \forall\ x,y \tag{8}$$

Even though the result of Equations 6 and 8 are mathematically identical, there is a subtle difference in the way they are computed. In Equation 6, a $kernelSize$ patch is read from $\mathcal{C}^\star$ and T, and written back on $\mathcal{C}^\star$ after the addition of $W \circledast P_i$; i.e., the convolution is applied locally. In Equation 8, only one value is read, processed and stored from/to $I^\star$ and T, but the convolution is applied over the whole image reducing the amount of per-event computation by $kernelSize^2$ times.