Towards Parallel Data Stream Processing on System-on-Chip CPU+GPU Devices

Gabriele Mencagli
Department of Computer Science
University of Pisa, Italy
gabriele.mencagli@unipi.it

Dalvan Griebler
Pontifícia Universidade Católica
do Rio Grande do Sul
Porto Alegre, Brazil
dalvan.griebler@pucrs.br

Marco Danelutto
Department of Computer Science
University of Pisa, Italy
marco.danelutto@unipi.it

Abstract—**Data Stream Processing is a pervasive computing paradigm with a wide spectrum of applications. Traditional streaming systems exploit the processing capabilities provided by homogeneous Clusters and Clouds. Due to the transition to streaming systems suitable for IoT/Edge environments, there has been the urgent need of new streaming frameworks and tools tailored for embedded platforms, often available as System-on-Chips composed of a small multicore CPU and an integrated on-chip GPU. Exploiting this hybrid hardware requires special care in the runtime system design. In this paper, we discuss the support provided by the WindFlow library, showing its design principles and its effectiveness on the NVIDIA Jetson Nano board.**

Index Terms—**Data Stream Processing, GPU Programming, CUDA, Multicores, System-on-Chips**

I. Introduction

Data Stream Processing (DSP) is a computing paradigm enabling the real-time processing of huge amount of information available as continuous data streams. An increasing number of scenarios can be modeled through the DSP paradigm, where streams are continuously transformed in order to extract insights, hidden knowledge and statistics reported to the end users. The continuous nature of data streams demands efficient systems for stream processing [1], which shall provide satisfactory performance levels to the end users.

Streaming systems have evolved from first-generation ones, supporting relational-algebra streaming tasks, to more general-purpose tools targeting scale-out architectures like Clusters and Clouds. With the advent of Edge computing platforms, parts of streaming workloads have moves closer to data producers, by avoiding the cost of transferring data to and back from the Cloud. However, supporting Edge/IoT resources requires special care because they are equipped with low-power architectures often based on System-on-Chips (SoCs).

The goal of this paper is to extend the C++17 WINDFLOW streaming library [2] to support embedded architectures composed of CPU+GPU SoCs. We target the family of boards (Tegra) shipped by NVIDIA. The scientific contributions of this paper are the following:

- we show a new run-time system based on efficient CUDA kernels supporting the processing of both stateless and stateful streaming transformations (operators);
- some optimizations are presented and discussed in relation to the features of the considered SoCs (use of pinned and unified memory, and prefetching directives);
- an experimental evaluation of WINDFLOW with different configurations, as well as a comparison with Apache Flink, a popular scale-out DSP system.

The paper organization is the following. §II introduces the background concepts. §III shows the preliminaries on the WINDFLOW library and its structured run-time system. §IV shows the paper contribution with the GPU support. §V presents the experimental evaluation, while §VI summarizes the related works. Finally, §VII draws the conclusions.

II. Background

DSP applications are Directed Acyclic Graphs (DAGs) of operators performing intermediate data transformation stages. Operators receive streams of records called *tuples*, and apply a processing function on each input by emitting results to other operators. They can be classified in two broad categories:

1) *stateless operators* apply a pure function on each tuple to produce the corresponding output result. The computed output solely depends on the given input;
2) *stateful operators* apply a logic that uses the current input and, in addition, an internal state to compute the corresponding result. The state is used to keep an history of what has been received so far by the operator.

Operators can be internally replicated to process different inputs in parallel. This parallelization can easily be exploited by stateless operators, where each input can be processed independently from the others. For stateful operators, the replication must respect the computation semantics. The most common pattern is the one of having a *partitioned state* [3]. Each input tuple has a special *key attribute* (a user-defined data type), and the processing on each tuple reads and modifies only the corresponding state partition associated with that key.

The parallelization of this computational pattern requires a *keyby distribution* of the input tuples to the replicas of the destination operator, in such a way that all the tuples having

the same key attribute are delivered to the same replica which keeps the corresponding state partition.

III. WINDFLOW LIBRARY

In this section, we recall the basic features of the WindFlow API and its runtime system (shortly, *runtime*).

A. API

The first step to create an application is to define a streaming environment as an instance of the `PipeGraph` class (see Fig. 1). The user can specify some configuration parameters related to the execution mode (e.g., with ordered timestamps or non-deterministic), and the timestamp creation policy.

```
PipeGraph app("myApp", Execution_Mode_t::DEFAULT,
                       Time_Policy_t::INGRESS_TIME);
```

Fig. 1: PipeGraph with timestamps assigned by Sources (i.e., ingress time) and non-deterministic mode (i.e., *default*).

Operators are created by leveraging *builder* classes having a fluent interface. The builders are instantiated with the processing logic of the operator (e.g., through a user-defined function, lambda, or a functor). Fig. 2 shows the creation of a *Filter* operator using a Boolean predicate to decide which tuples (of type `tuple_t`) to drop. The Filter is created with three internal replicas to improve throughput. In the example, the Filter is stateful: each replica maintains a private hash table mapping keys onto state objects, which are used to keep statistics of the received tuples with the same key. To enable this behavior, the operator is configured to process inputs in a *keyby* manner, using a key extractor (a lambda in the example) to extract the key attribute of type `key_t` from the tuple.

```
struct Filter_Functor { bool operator()(tuple_t &t){…}};

Filter_Functor myfunctor;
Filter filter = Filter_Builder(myfunctor)
  .withParallelism(3)
  .withName("MyFilter")
  .withKeyBy([](const tuple_t &t) -> key_t { return t.key; })
  .build();
```

Fig. 2: Filter with keyby processing and three internal replicas.

Once created, the operators are connected in the right order as in Fig. 3. In the example, we have a three-staged logical pipeline with a Source, the Filter, and a Sink operator.

```
app.add_source(source).add(filter).add_sink(sink);
app.run();
```

Fig. 3: Interconnection of operators.

B. Structured Runtime

The runtime is built using a composition of *building blocks* [2] (shortly, *blocks*), which can be nested and composed to create the application structure. Sequential blocks are:

1) *wrapper node*, or simply *node*, is a block encapsulating a user-defined function applied to each received tuple in order to produce the corresponding output result;

2) *combiner* is a block built on top of two nodes or combiners. Given two nodes N_1 and N_2 with processing functions F_1 and F_2, a combiner block executes the composition of the two functions $F_2(F_1(x))$ on each input x. By combining combiners with combiners, we can fuse an arbitrarily long chain of transformations.

Parallel blocks express communications between nodes and combiners, or between parallel blocks themselves. They are:

1) *pipeline*: given a set of sequential blocks connected in series, the pipeline expresses temporal parallelism, where the blocks work in parallel on different inputs;

2) *all-to-all* (A2A): it is composed by two sets of sequential or parallel blocks, the left-hand set (LS) and the right-hand set (RS). Each block in the LS is connected to all the blocks in the RS.

The application in §III-A is implemented by composition of the blocks in Fig. 4. Nodes performing the operator processing functions are combined with nodes performing run-time support activities: *emitters* (E) perform the distribution of tuples to the replicas of the next operator, while *collectors* (C) perform the multiplexing of tuples received from the different replicas of the preceding operator. The presence of stateful operators (the Filter in the example) requires the use of a A2A block to connect all the replicas of the Source with the ones of the Filter. The whole structure is a tree of nested blocks.
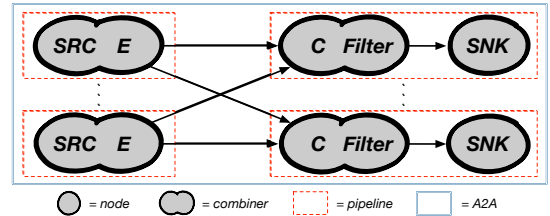

Fig. 4: Structure of the example using building blocks.

WINDFLOW is based on the FASTFLOW building blocks [4]. The characteristics of this implementation are:

- each node is executed by a dedicated thread, except nodes incorporated into the same combiner block, which are executed by the same thread sequentially;
- communication channels are implemented by single-producer single-consumer lock-free queues of memory pointers to heap-allocated data structures.

IV. ADDING GPU SUPPORT TO WINDFLOW

The new set of GPU-based operators added to the library exchange tuples in batches. The batch size is set with the `withOutputBatchSize()` method of the builder. The builder constructor takes as input argument the user-defined functional logic, which is provided through a `__device__` lambda or a `operator()` method of a functor object. We support two GPU-based operators working in a stateless or stateful mode: `Filter_GPU` able to drop inputs based on a Boolean predicate; `Map_GPU` able to execute a user-defined function producing one output per input.

A. Stateless Operators

In the stateless version, each tuple can be processed in parallel with respect to the others. The source code of the CUDA kernel is shown in Fig. 5 for the Map_GPU operator. Each CUDA thread of the kernel works on a specific input of the batch by calling the user-defined functional logic of the operator. In case of the Map_GPU, the logic modifies the input in place. For the Filter_GPU operator (whose code is not shown for brevity), the logic returns a Boolean flag.

```
template<typename tuple_t, typename map_func_t>
__global__ void stateless_kernel(batch_item_t<tuple_t> *data_gpu,
                                 size_t len,
                                 int num_active_thread_per_warp,
                                 map_func_t func_gpu) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = gridDim.x * blockDim.x;
    int threads_per_group = warpSize / num_active_thread_per_warp;
    int num_workgroups = num_threads / threads_per_group;
    int id_group = id / threads_per_group;
    if (id % threads_per_group == 0) {
        for (size_t i= id_group; i<len; i+= num_workgroups)
            func_gpu(data_gpu[i].tuple);
    }
}
```

Fig. 5: CUDA kernel for the Map_GPU stateless operator.

Since the user-defined logic (func_gpu) can be general, threads belonging to the same warp can follow different conditional paths. To partially mitigate this issue, the kernel is configured to use only warp-level parallelism if the batch size is smaller than the maximum number of resident warps on the device. This is achieved by activating one thread per warp in that case, while the others remain idle. When the size of the batch is bigger, we activate more threads per warp reaching the full utilization of all the resident CUDA threads in case of large batches. For the Filter_GPU operator, the kernel executes in parallel the func_gpu logic by obtaining an array of flags, and then the host thread of the replica calls a thrust::copy_if() to compact the batch.

B. Stateful Operators

Stateful operators keep an internal state partitioned by key. A batch is composed of several inputs that may belong to different keys. In our approach, the state partition associated with a key attribute is an object fully resident in the GPU-accessible memory. The user-defined logic takes an input tuple and the state object of its key, and modifies the tuple by accessing the corresponding state partition (Filter_GPU returns a Boolean flag instead). Each batch structure is extended with the following support arrays:

- start_idxs is a GPU array of integers of size equal to the number of distinct keys present in the batch. The element start_idxs[i] contains the position in the batch of the first tuple having the i-th key;
- map_idxs is a GPU array of integers of size equal to the batch size. The element map_idxs[i] contains the position in the batch of the next tuple having the same key of the i-th element (or -1 if it does not exist).

Fig. 6 shows an example of the support arrays in case of a batch of eight tuples with three keys A, B, C.
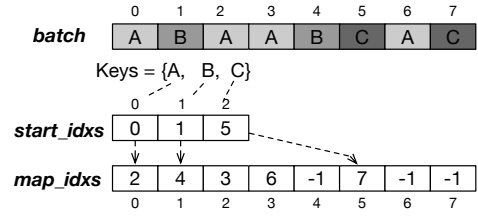

Fig. 6: Support arrays with a batch of 8 tuples and 3 keys.

The kernel is described in Fig. 7. Each key is assigned to a CUDA thread in charge of computing the user-defined function over all the tuples of the batch having that key.

```
template<typename tuple_t, typename state_t, typename map_func_t>
__global__ void stateful_kernel(batch_item_t<tuple_t> * data_gpu,
                                int *map_idxs,
                                int *start_idxs,
                                state_t **states_ptr,
                                int num_keys,
                                int num_active_thread_per_warp,
                                map_func_t func_gpu)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = gridDim.x * blockDim.x;
    int threads_per_worker = warpSize / num_active_thread_per_warp;
    int num_workgroups = num_threads / threads_per_group;
    int id_group = id / threads_per_group;
    if (id % threads_per_group == 0) {
        for (int id_key=id_group; id_key<num_keys; id_key+= num_workgroups) {
            size_t idx = start_idxs[id_key];
            while (idx != -1) {
                func_gpu(data_gpu[idx].tuple, *(states_ptr[id_key]));
                idx = map_idxs[idx];
            }
        }
    }
}
```

Fig. 7: CUDA kernel for the Map_GPU stateful operator.

The kernel uses an additional array states_ptr, which contains at position i a pointer to the state object corresponding to the i-th key of the batch. While state objects are used by the CUDA threads in the kernel, their creation is done by the host threads running the replicas of the GPU-based operator on the CPU. Each host thread, once a batch is received, iterates across all the distinct keys of the batch, and allocates the state objects each time a key is seen for the first time. In order to create the state of a key once, we use the Intel TBB concurrent hash table to map key attributes onto pointers in GPU-accessible memory to the corresponding state objects.

The preparation of the states_ptr array, and the access to the hash table is done is parallel by the replicas on different batches. To avoid more CUDA threads (e.g., of simultaneously launched kernels) work on the same state partition in parallel, at most one replica of the same operator can execute the stateful kernel at the same time through the use of a lock. The preparation of the two support arrays map_idxs and start_idxs is done by a specific emitter node functionality.

C. Advanced Implementation Choices

Further important aspects have been addressed in addition to the implementation choices already described before.

1) Use of CUDA streams: to avoid synchronization, we use CUDA streams to execute kernels in parallel on the device and to synchronize only when required in order to produce correct results. Each batch is associated with its own CUDA stream, which is used by all the host threads that work on that batch.

2) Recycling GPU memory buffers: allocating GPU-accessible buffers is a costly operation. We use a *recycling mechanism* of batches, where each batch incorporates a pointer to a multi-producer single-consumer lock-free queue. Operator replicas, where batches are consumed, do not deallocate batches in general, but push pointers to batches into the corresponding queue. Emitters in charge of creating batches do not allocate them each time, but try to pop a pointer from their recycling queue in order to reuse already allocated buffers.

3) Implementation variants: due to the specific features of Tegra SoC devices, several variants have been designed:

- *Explicit transfers*: this variant adopts separated buffers for storing GPU-accessible data, and copies from CPU to GPU buffers are done explicitly and possibly overlapped;
- *Unified memory*: this variant adopts unified buffers by the host and the GPU. On Tegra devices, the physical memory is shared by the CPU and GPU, and the use of unified memory avoids useless copies. Furthermore, it makes the caches (both on CPU and GPU) coherent;
- *Pinned memory*: we use pinned memory allocated with `cudaMallocHost()`, because these buffers are shared by the CPU and the GPU. However, pinned memory is not cached on the CPU side (in SoCs with compute capability lower than `7.2` like the Jetson Nano).

The version with unified memory can be extended with the use of explicit prefetching directives, which are implemented by the `cudaStreamAttachMemAsync()` calls.

V. Experiments

We provide an evaluation of the performance achieved by WINDFLOW with the new GPU support presented in this paper. The reference hardware is the NVIDIA Jetson Nano board. It is a SoC composed of a quad-core ARM Cortex-A57 MPCore processor working at maximum frequency of 1.4GHz, and a Maxwell GPU with 128 CUDA cores. The board is equipped with 4GB of RAM and 32GB of storage.

We use two applications provided in the DSPBench benchmarks [5]. The first, SpikeDetection (SD), finds out spikes in a stream of sensor readings. The second, FraudDetection (FD), applies a Markov model to discover credit card frauds. The two DAGs are pipelines of four and three operators. For SD, we have a Source, a stateful Map (Moving Average), a stateless Filter (Spike Detector) and a Sink. For FD, the pipeline has three operators: a Source, a stateful Filter (Predictor) and a Sink. All the experiments have been compiled with `clang` version `v10.0` (with the `-O3` flag) and CUDA `v10.2`.

1) Performance on the ARM CPU: the first evaluation has been done using CPU-based operators only. In order to use the four CPU cores at best, all the operators are configured to use two replicas, and we chain consecutive operators not connected by a keyby distribution to reduce the amount of threads. So doing, each application is run by 4 host threads, each pinned on a CPU core. Fig. 8 reports the results which are the average of tens repetitions (error bars are not shown for brevity, since they are small).
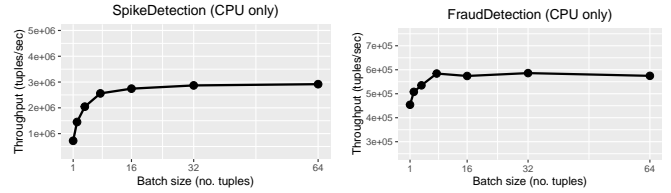

Fig. 8: Throughput of SpikeDetection (left) and FraudDetection (right) using WindFlow with CPU-only operators.

In the experiments, we vary the size of the batches. Since we are not using the GPU, batches of few tens of inputs are sufficient to amortize the run-time system overheads. For SD, we measure 2.7M inputs consumed per second with batches of 16 tuples each. Greater batches do not improve this value. For FD, the peak throughput is of 580K inputs per second with slightly smaller batches of eight tuples each. We can observe that SD is a finer-grained application than FD, since its peak throughput is about five times higher.

2) Performance with CPU-GPU: we repeat the experiments with our GPU support, where all the operators (except Sources and Sinks) use the GPU. The size of the batches is much bigger, up to thousands of tuples per batch. Fig. 9 shows the results, which raise the following comments:

- the version with pinned memory is the one having the lowest runtime overheads, since no copies and no coherency activities by the CUDA runtime are required. However, on Tegra devices pinned memory is not cached by the GPU, and on Jetson Nano it is not cached by the CPU too. This hardware limitation poses serious performance issues on FD, since the predictor computation has a large working set which cannot be stored in cache.
- the version with explicit transfers performs reasonably well, outperforming the others for FD. Although this version copies batches from CPU to GPU explicitly, copies are overlapped and their cost is well masked;
- unified memory, although its potential in avoiding useless copies, requires additional coherency and cache maintenance operations that are costly. The use of prefetching directives only alleviates this problem.

The gain with respect of using only the CPU is of 3.17x for SD and 2.12x for FD.

3) Impact of recycling GPU buffers: we repeated the tests without the recycling support described in §IV-C2. The throughput drops of 20% and 17% for SD and FD respectively.

4) Comparison with standard tools: we conclude with a comparison against Apache Flink (`v1.9.0`) compiled with Java `v11.0` and using only the ARM CPU, since GPUs are not natively supported by Flink. Fig. 10 shows the results in the best configuration found with the peak throughput. As we can observe, WINDFLOW outperforms Flink with a 11x higher throughput with SD (6x with FD).

VI. Related Works

Making the DSP paradigm and its frameworks suitable for IoT and Edge is a hot research topic. Some works focus on the
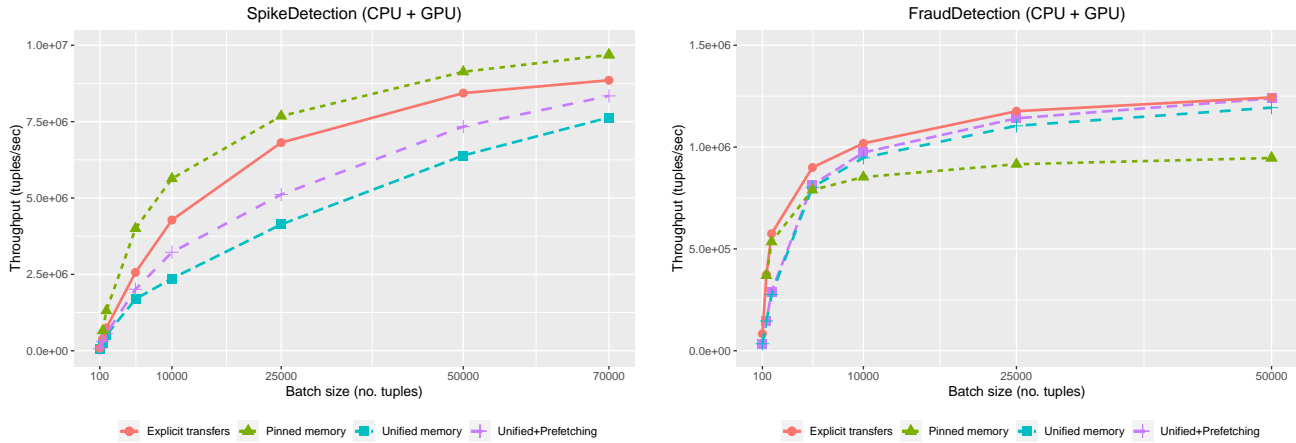
Fig. 9: Throughput of SpikeDetection (left) and FraudDetection (right) using WindFlow with CPU+GPU processing.
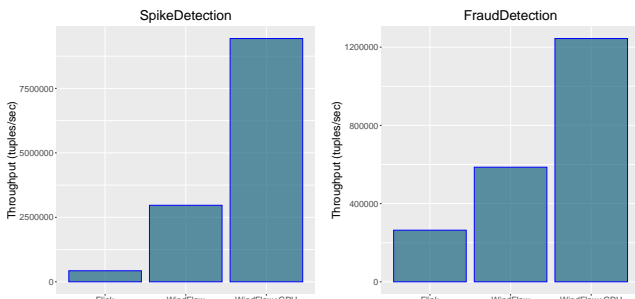

Fig. 10: Comparison against Apache Flink.

scalability of run-time systems on single multicores [6]. They trade off scalability with latency by dynamically scheduling the processing of input batches on a pool of threads. However, these systems do not support accelerators, which are a valuable component for SoC-based IoT/Edge resources.

EdgeWise [7] is a recent work enhancing Apache Storm for edge resources. They propose a model where operators are logical executors dynamically scheduled onto a fixed-size pool of threads according to application-dependent scheduling policies. On GPUs, the pioneering work is Saber [8], which provides streaming support on GPU for relational algebra operators. While interesting, this system does not support user-defined stateful operators like the ones described in this paper. Gasser [9] focuses on GPU-based streaming operators working with sliding windows, so for a very special class of operators only. FineStream [10] extends the work done with Saber (so for relational-algebra queries only) on integrated GPUs. Although with some points in common with our work, the source code of FineStream is not publicly available.

VII. CONCLUSIONS

This paper presented the extension of the WINDFLOW library for NVIDIA SoCs. The implementation choices in the run-time system design showed a clear benefit in using the on-chip GPU. Furthermore, the comparison between the proposed implementation variants produced interesting

insights for future extensions of this work, in order to use dynamically the best implementation variant based on the workload characteristics of the application and batch size.

REFERENCES

[1] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2651–2658. [Online]. Available: https://doi.org/10.1145/3318464.3383131

[2] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi, and M. Danelutto, "Windflow: High-speed continuous stream processing with parallel building blocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2748–2763, 2021.

[3] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, p. 517–539, Aug. 2014. [Online]. Available: https://doi.org/10.1007/s00778-013-0335-9

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *FastFlow: High-Level and Efficient Streaming on Multicore*. John Wiley & Sons, Ltd, 2017, ch. 13, pp. 261–280. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13

[5] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.

[6] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "Lightsaber: Efficient window aggregation on multi-core processors," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2505–2521. [Online]. Available: https://doi.org/10.1145/3318464.3389753

[7] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 929–946. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fu

[8] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 555–569. [Online]. Available: https://doi.org/10.1145/2882903.2882906

[9] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

[10] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 633–647.