[DRAFT] Online Tracking Algorithms on GPUs for the $\overline{\text{P}}$ANDA Experiment at FAIR

[FIXME: Author list to be completed according to publisher's rules]

Abstract. $\overline{\text{P}}$ANDA is a future hadron and nuclear physics experiment at the FAIR facility in construction in Darmstadt, Germany. In contrast with the majority of current experiments, $\overline{\text{P}}$ANDA's strategy for data acquisition is based on event reconstruction and selection from free-streaming data, performed in real time entirely by software algorithms using global detector information. This paper reports the status of the development of algorithms for the reconstruction of charged particle tracks, optimized for use in online data processing applications, using General-Purpose Graphic Processing Units. Two algorithms for trackfinding, the Triplet Finder and the Circle Hough are described, with details of their GPU implementations. Average track reconstruction times of less than 100 ns are obtained running the Triplet Finder on state-of-the-art computing-grade GPU cards. In addition, a proof-of-concept system for the dispatch of data to tracking algorithms using Message Queues is presented.

1. Introduction

1.1. The $\bar{P}$ANDA Experiment

The $\overline{\text{P}}$ANDA (Anti-**P**roton **An**nihilation at **Da**rmstadt) experiment is one of the main experiments at the FAIR (**F**acility for **A**ntiproton and **I**on **R**esearch in Europe) facility, currently in construction in Darmstadt, Germany. $\overline{\text{P}}$ANDA will study collisions between a cooled antiproton beam ($\Delta p/p \sim 10^{-5}$) and a fixed proton target, in the momentum range 1–15 GeV/c. The unique $p\bar{p}$ initial-state configuration allows $\overline{\text{P}}$ANDA to study a wide range of research topics in the area of Quantum Chromodynamics (QCD) including hadron spectroscopy, hypernuclei, and nuclear structure [**?**].

$\overline{\text{P}}$ANDA shares with many other experiments the challenge of studying rare physics processes in a collision environment dominated by background events. The high luminosity needed to accumulate sufficient statistics means that an enormous amount of uninteresting background events, whose cross-section is many order of magnitude greater than the signal, is produced as well. The total data rate coming from the detector greatly exceeds the capabilities of offline storage, so some form of online signal/background rejection needs to be implemented.

The conventional strategy to reduce the incoming data is to perform a fast, low-level trigger selection, implemented in hardware, based on information from a subset of the detector. If the outcome of the decision is positive, data from the whole detector are collected and stored offline for further analysis.

In the case of $\overline{\text{P}}$ANDA, this approach is not feasible. Due to the nature of hadronic interactions in the energy range of interest, background and signal events are very similar, and information from the full reconstructed event is needed to perform the selection. Additionally, the antiproton beam used by $\overline{\text{P}}$ANDA is designed for quasi-continuous operation, in opposition to the synchronous, bunched beam structure available at collider experiments. $\overline{\text{P}}$ANDA's strategy for data acquisition

is instead based on an online event filtering scheme. Data from the whole detector is read continuously. An initial stage of event building is performed by FPGA-based compute nodes. Then, software algorithms perform full reconstruction of each event and background/signal identification based on multiple software trigger lines. Events passing the trigger selection are saved for offline storage. The incoming data rate from the detectors is approximately 200 GB/s. To match the available offline storage capabilities of 3 PB/year, an online background rejection factor of about 1000 is required.

1.2. The $\bar{P}$ANDA Central Tracking Detectors

The $\overline{\text{P}}$ANDA detector complex can be divided into two main sections, the Central Detector and the Forward Detector. In the Central Detector, the focus of this paper, A 2T solenoidal magnetic field bends the trajectory of charged particles into three-dimensional helixes, with their axis parallel to the beam direction. The two-dimensional projection of tracks on the transverse plane are thus circles. The tracking detector closest to the beam is the Micro Vertex Detector (MVD). The MVD is a solid-state detector composed of a combination of pixel detectors (10.3×10^6 channels) and double-sided silicon strips (2×10^5 channels), capable of a vertex resolution of < 100 μm. Surrounding the MVD is the Straw Tube Tracker (STT). The STT is made up of 4636 drift tubes, or straws, filled with a 9:1 Ar/CO_2 gas mixture, and it has a spatial resolution $\sigma_{xy} \sim 150$ μm, $\sigma_z \sim$ 2–3 mm. Three Gas Electron Multiplier (GEM) disks are located in the forward direction, with about 35000 readout channels and a spatial resolution of < 100 μm.

1.3. Online Tracking on GPUs

Online tracking is an essential step in the $\overline{\text{P}}$ANDA DAQ chain. Online tracking algorithms take in input discrete hit data from the tracking detectors, and process them to produce in output continuous particle trajectories, or tracks. The information from the number and the features of each track is used as input to the algorithms for online event building and event selections.

A Graphic Processing Unit (GPU) is a type of hardware architecture, designed for data-parallel computation. Originally created for accelerating graphic-intensive application in personal computers, thanks to the introduction of high-level programming paradigms such as CUDA or OpenCL and the release of dedicated hardware, GPU-based processors are increasingly used for general computation, a paradigm known as General-Purpose GPU (GPGPU). For computational tasks that can be formulated in a data-parallel fashion, using GPUs in alternative to CPUs can result in a significant increase in performance, in terms of both absolute and relative performance, i.e. considering the computational power per units of cost of hardware and power consumption. However, not all applications are well-suited for porting to the GPU, since a high level of parallelism is indispensable to reach the full potential of the GPU. In addition, the different hardware architecture results in programming strategies that can be radically different from their CPU counterparts.

Online particle tracking on GPUs is being employed in many experiment in high-energy and nuclear physics. Tracks can be processed independently from each other, so trackfinding algorithm have at least one intrinsic level of parallelism. Furthermore, the data sets are generally small, resulting in high arithmetic intensity, i.e. the ratio of computation to memory transfer. Most notably ATLAS [**?**], CMS [**?**] and ALICE [**?**] at LHC all make use of GPU-based algorithms in their High-Level Trigger stage of the DAQ chain. The goal of $\overline{\text{P}}$ANDA is, however, to include GPU-based online track reconstruction at an earlier stage of the online reconstruction chain, without a previous stage of hardware trigger selection.

2. Triplet Finder Algorithm

2.1. Algorithm Concept

The Triplet Finder algorithm is a track finding algorithm, developed specifically for the $\overline{\text{P}}$ANDA STT detector. The core idea of the Triplet Finder revolves around the fast reconstruction of a circle based on a restricted subset of hit combinations. Three sets of pivot straws are arranged in contiguous configuration called pivot layers, as illustrated in Figure 2. As soon as a hit is detected in a pivot straw, the neighboring straws are checked for hits. A virtual hit called triplet is then defined as the center of mass of the three hit points. Once two triplets have been calculated, a circular track is computed from the two triplets and the interaction point in $(0, 0)$. The last step of the algorithm consists of associating remaining hits lying on the trajectory of the circle with the track.

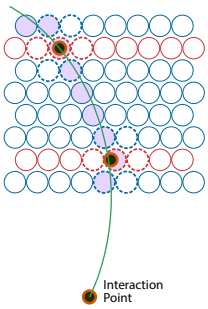

Figure 1: Sketch of the principle of operation of the Triplet Finder algorithm.

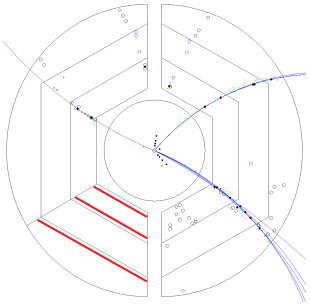

Figure 2: Schematic section of the STT in the (x, y) plane. Pivot layers are shown for one of the six symmetrical straw sectors. Triplets (orange-black dots) and reconstructed tracks (blue) are also shown.

2.2. GPU Implementation

The GPU version of the Triplet Finder algorithm is implemented using the CUDA programming language [**?**]. In this section, techniques and improvements of more general applicability are highlighted; for a detailed description of the GPU implementation of the algorithm, see [**?**].

Bunching The main challenge in the GPU implementation of the TF algorithm comes from its computational complexity of $\mathcal{O}(N^2)$, where N is the number of hits processed at the same time. This drawback is encountered solely in the GPU version, where $\mathcal{O}(10^4)$ hits should be processed in parallel to ensure full utilization of the GPU. One strategy to circumvent the complexity is to subdivide hits in sets, called *bunches*, that can be processed indipendently. This restricts the $\mathcal{O}(N^2)$ complexity to the number of hits in each bunch, while the full occupancy of the GPU can still be ensured by running multiple bunches in parallel. The subdivision is implemented as follows. First, the simulation is divided in segments of duration T_C, the core time. An additional time T_D, corresponding to the maximum straw tube drift time, is added at the end of each bunch. Hits are then assigned to bunches on the basis of their timestamp t_0. If N_C and N_D are the average number of hits in the core time and the drift time, respectively, the resulting complexity is then $\mathcal{O}(N\frac{(N_\text{C}+N_\text{D})^2}{N_\text{C}})$, which is linear in the total number of hits N. Duplicate tracks can occur as a result of partially overlapping bunches. This can be addressed by a later track coalescing stage. Figure 4 shows the effects of the adoption of bunching. It is the most valuable improvement in terms of performance for the GPU version of the Triplet Finder algorithm.

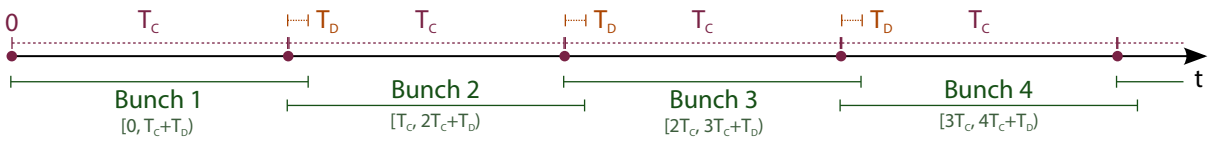

Figure 3: Scheme of bunching.

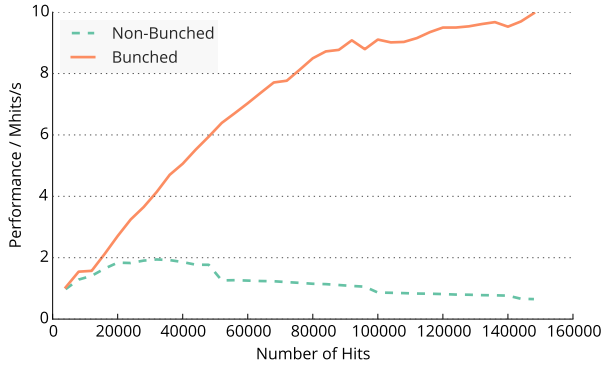

Figure 4: Comparison of the effect of bunching in terms of performance (number of hits processed per unit time) as a function of the total number of hits.

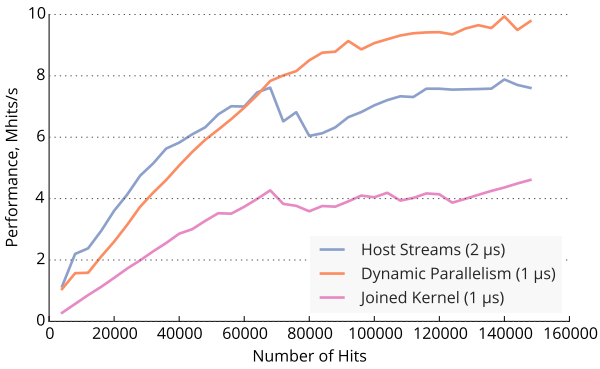

Figure 5: Performance as a function of the total number of processed hits for different kernel dispatch strategies.

Strategies for bunch dispatch on the GPU Three different strategies for dispatching multiple bunches on the GPU have been tested. The *dynamic parallelism* approach makes use of nested kernels. The host launches one master kernel, with one thread for each bunch. Each master kernel instance then launches separate individual slave kernels for each phase of the algorithm, with one thread per hit. In the *host streams* approach, individual kernels for each kernel are similarly used, but they are launched by the host, with one CUDA stream per bunch. In the *joined kernel* approach, all phases of the algorithm are performed within a single kernel. The host invokes the joined kernel with one thread block for each bunch. A summary scheme of the various bunch dispatch strategies, along with a comparison of their performance, can be seen in Figure 5. For a high number of processed hits, the best performance results from the dynamic parallelism approach.

Effect of Data Packing Ensuring good memory access patterns is particularly important for GPU applications, and more efficient memory access patterns almost always result in improved performance. In the case of Triplet finder, a performance improvement of up to 20% can be reached when using Structures of Arrays (SoA) instead of the more intuitive Arrays of Structures (AoS).

Comparison of server-grade and consumer cards GPU-based devices currently available on the market can be divided in two broad categories: consumer-grade cards, designed for accelerating graphics on personal computers, and server-grade cards, built specifically for use in High-Performance Computing. Although it's not their primary purpose, recent consumer-grade cards are able to run CUDA applications. Their inferior performance in absolute terms w.r.t. server-grade cards is accompanied by a much lower retail price. As an indicative comparison, the same version of Triplet Finder algorithm is run for one server-grade card, NVIDIA Tesla K20X, with 2688 CUDA cores, a peak single-precision performance of 3950 GFLOPS and a price of about

3200 EUR, and a consumer-grade card, NVIDIA GeForce GTX 750 Ti, with 640 CUDA cores, a peak single-precision performance of 1306 GFLOPS and a price of about 140 USD. The K20X has a peak performance for the Triplet Finder algorithm of about 10 Mhits/s, to be compared to the 5 Mhits/s for the GTX 750 Ti. In terms of Mhits/s/USD, the consumer-grade card has about 10 times more the price-normalized performance w.r.t. the K20X. However, further studies are needed to quantify how the drawbacks of using consumer-grade cards, e.g. the lack of support of ECC memory correction features, and the absence of vendor technical support, can affect reliability and data integrity in an online data processing scenario.

2.3. Summary and Outlook

Taking into account all optimizations and pre-processing steps, the Triplet Finder algorithm processes 8.3 Mhits/s on a Tesla K20X card. On a Tesla K40X, a more modern card overclocked with the GPUBoost technology, the performance is 10.3 Mhits/s. Although the GPU version of the algorithm can be considered deeply optimized, a number of possible future improvements are identified, including the porting to the GPU of the pre-processing phases currently being performed on the CPU, and the introduction of hit bookkeeping to reduce the computational complexity of the hit association phase. A complete description of the Triplet Finder algorithm can be found in [**?**, **?**]. A detailed description of the GPU implementation is available on [**?**].

3. Circle Hough Transform

The Circle Hough algorithm is a novel algorithm for trackfinding, developed at $\overline{\text{P}}$ANDA during the last year. Compared to the Triplet Finder, it is not exclusive to the $\overline{\text{P}}$ANDA STT. In the current version, it can be applied to hits originated in all central tracking detectors: MVD, STT, and GEM.

3.1. Algorithm Concept

The Circle Hough algorithm is based on the principle of the Hough transform, a technique for feature extraction used e.g. for detecting straight edges in a picture. In this method, an input dataset is checked against a model, described by a number of parameters. The goal is to find the set of parameters corresponding to the model that fits best with the input data. A family of prototype models is generated for each point of the input dataset; the resulting parameters are collected and the resulting parameter space, the Hough space, is analyzed. Points in the Hough space with the highest density, corresponding to the parametres occurring with the highest frequency in the input dataset, coincide with the model that fits best to the data.

In the case of track finding, the input dataset consists of hit points, and the feature to be extracted are tracks that fit best to the hit points. In the Circle Hough algorithm, tracks are represented as circles in the xy plane, passing through both the interaction point in (0,0) (IP) and the hit point in $(x_{\text{hit}}, y_{\text{hit}})$. A circle satisfying these requirements is referred to as *Hough circle*. The condition of passing through the hit point is valid for point-like hit points, such as hits originated in the MVD or the GEMs. For extended hit points, such hits in the STT, Hough circles must be tangent to the circle centered in $(x_{\text{hit}}, y_{\text{hit}})$, whose radius is r_{iso}, the isochrone radius, calculated from the drift time of the hit. For each hit point, all possible Hough circles are computed; this is equivalent to defining all possible tracks compatible with the hit point. The process is repeated for all hits, sampling the Hough space. Since Hough circles are uniquely described by two parameters, e.g. the coordinates of the circle center $(x_{\text{C}}, y_{\text{C}})$, the Hough space will be two-dimensional. The Hough space is then scanned for peaks; the $(x_{\text{C}}, y_{\text{C}})$ coordinates of the peaks correspond to the parameters of the tracks in the real space.

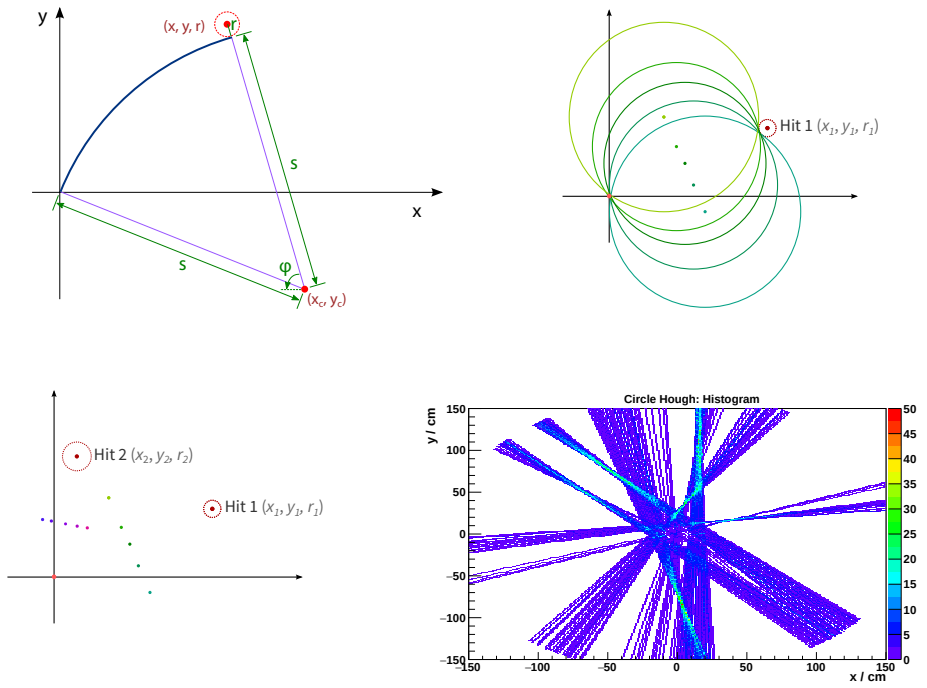

Figure 6: Schematic diagrams for the Circle Hough algorithm. From left to right, top to bottom: (1) Definition of angle φ for a Hough circle around the STT hit point in (x, y, r). (2) Several Hough circles are created. (3) The operation is repeated for other hit points. Some Hough circle centers are pictured. (4) Example of representation of the discretized 2D Hough space.

3.2. GPU Implementation

Altough the Circle Hough algorithm was originally tested on the CPU, its native degrees of parallelism, in both the hits and the Hough circle calculation for each hit, make the development of a GPU version a natural choice. The GPU version is implemented in CUDA C/C++, and tested on different NVIDIA cards. The computational phases of the algorithm can be outlined as follows. First, the sampling values, used as the input of the Hough circle generation, are defined. In the simplest approach, this is a constant set of values for the angle φ between the IP and the hit point in the range $[0°, 360°)$, with a fixed sampling granularity, $\Delta\varphi$, resulting in N_φ angles in total.

Then, the Hough circles are calculated. Hit data are copied on the device; then one kernel is invoked, with each thread computing one set of Hough circle center coordinates for each hit and each value of φ. Block size is determined to ensure the maximum occupancy of the GPU. Optimal results are reached with block sizes of 128–256.

The set of Hough circle center coordinates is copied back to the host, where the subsequent phases of the algorithm are performed. The Hough space is discretized in a 2D histograms, which is then filled with the Hough circle center coordinates. A peak-finding algorithm identifies peaks in the histogram, and their coordinates are then saved to extract track parameters.

3.3. Summary and Outlook

Analysis of the performance of the CUDA code show that the step of copying back the Hough circle center coordinates to the host is the current limiting factor. Even using pinned host memory, it accounts for 80% and 90% of the total runtime on an NVIDIA GeForce GTX 750 Ti and an NVIDIA Tesla K40X, respectively. The kernel-only performance of the Hough circle computation of the algorithm is 30 Mhits/s on a Tesla K40X. Development in the near is thus

targeted towards implementing on the GPU the phases of histogram filling and the subsequent peak-finding. The possibility to use ad-hoc sampling values for each hit instead than the constant set of angle values, and the adoption of more advanced schemes for the kernel dispatch are also being explored.
Detailed studies of the Circle Hough algorithm, including an analysis of the physics performance for a benchmark channel can be found in [**?**].

4. Use of Message Queues in Combination with GPUs

The development of a suitable communication infrastructure is one of the central challenges in the design of a complex system for data acquisition and processing. The communication layer must provide the optimal balance between performance, stability, and use of resource. Additionaly, it needs to provide a compatibility layer among all different types of hardware (CPUs, Many Integrated Cores (MIC) co-processors, GPUs, FPGAs) used for the data processing. Furthermore, it should be flexible enough to accomodate changes both in the architecture of the system, and the possible future evolutions of low-level data transport protocols and interfaces.
One possible approach, at least for later stages of the data processing chain, is to use Message Queues (MQ) as the basis for the exchange of data and control. In a MQ environment, both data and control packages (messages) are exchanged asynchronously between senders and receivers through a control structure (queue), making for an inherently flexible, easily scalable architecture. The actual data processing is distributed among different independent tasks, running in parallel on different machines, networks and processor types, exchanging data through the MQ.
FairMQ [1] is the implementation of MQs within the FairRoot framework. The main idea of FairMQ is to provide MQ functionality by means of a unified, high-level abstract interface. The same classes are used to create and access messages regardless of the type of communication (intra- or inter-process), the protocol, or the channel (PCIe, network, ...), providing a high degree of versatility by design. The choice of the underlying low-level library is also open, leaving the possibility to change it or to adopt optimized versions. At the moment, ZeroMQ and nanomsg are supported.

4.1. Circle Hough Test System

To explore the possibilities of using FairMQ in association with GPU-based tracking, a standalone, proof-of-concept system integrating FairMQ and an implementation of the Circle Hough algorithm described in Section 3 has been developed. The functionality of the system is subdivided into independent modules (Fig. 7), structured as follows. An instance of `FairMQDevice`, the *Sampler*, processes the input data and initiates the FairMQ stream. First, the `ReadFile` module to read hit data from an input file is called. The data are serialized into arrays, and copied to `FairMQMessage`s. The message transmission is managed by an istance of `FairMQTransportFactory`. A second instance of `FairMQDevice`, the *Receiver*, receives the FairMQ stream and performs the Circle Hough computations. First, the hit data is extracted from the `FairMQMessage`s and converted to the appropriate data structures. The Circle Hough transform calculations are performed either or the CPU or on the GPU by the the `CircleHoughCPU` and `CircleHoughGPU` modules, respectively. Lastly, the Hough circle center coordinates are collected by the `CreateHisto` module.
Each instance of FairMQDevice runs independently from the others, and automatically initiates the data processing as soon as a FairMQ communication channel is established. Once the system is properly set up, alterations to the architectures of the system, e.g. by adding processing nodes or changing the communication channel, can be done at runtime, with no alteration to the source code.

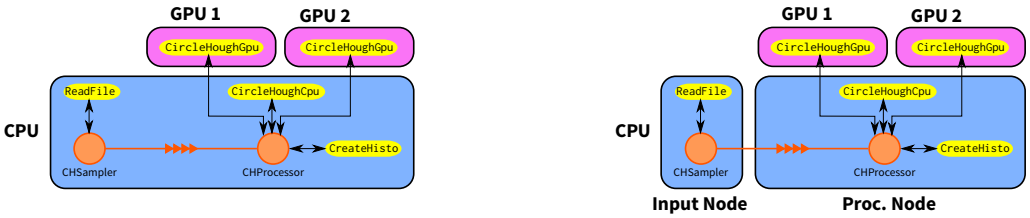

Figure 7: Scheme of the system. The orange circles represent instances of `FairMQDevice`, with the orange arrows representing the transmission of FairMQ messages within the same machine (left), and between two machines over a network (right).

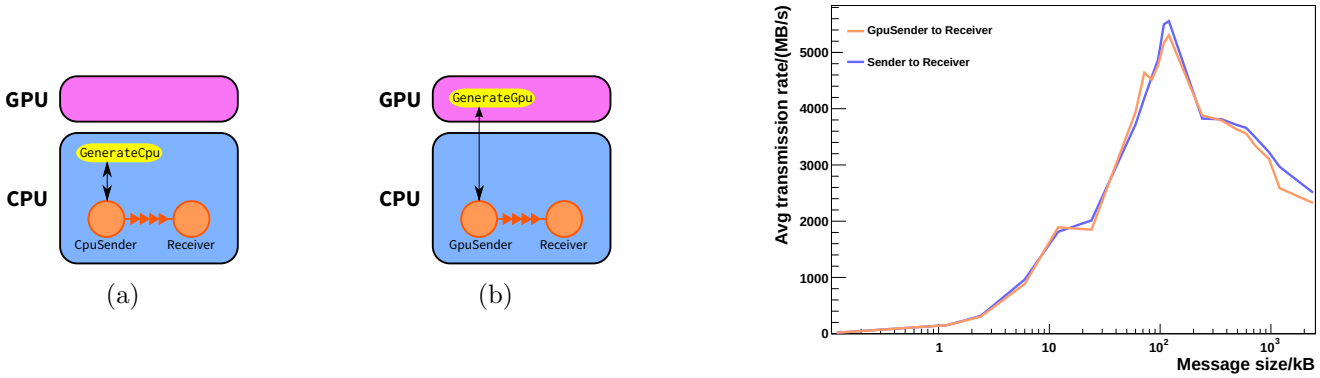

Figure 8: Scheme of the system with no computation when the transmission data packet is generated (a) on the CPU, and (b) on the GPU. The orange circles represent instances of `FairMQDevice`, with the orange arrows representing the transmission of FairMQ messages. (c) Average transmission rate as a function of the message size for the two configurations.

Transmission Rate Test The transmission rate in FairMQ depends on many factors, such as the transport library, the communication protocols, and the communication channel. A bare-bones system to study the transmission rate without any overhead from computation consists of two `FairMQDevices`, a *Sender* and a *Receiver*, running on the same machine. A data packet is generated once either on the CPU or on the GPU, and messages are sent continuously over a socket using the TCP/IP protocol. The size of the message is varied and the time-averaged transmission rate is measured from the FairMQ logging functionality. Fig. 8 shows that a transmission rate of over 5 GB/s is reached for a message size of about 100 kB. While these results should not be considered as a formal benchmark of FairMQ, they are indicative of the fact that the maximum achievable is limited by the low-level data transport infrastructure rather than by FairMQ itself.

4.2. Summary and Outlook

The test system described in this section constitutes a first look into the inclusion of FairMQ with GPU-based tracking algorithms. Future work is aimed at its integration with the PandaRoot framework, eventually evolving into a prototype of a full-featured implementation of a online reconstruction chain.

[1] M. Al-Turany, D. Klein, A. Manafov, A. Rybalchenko, and F. Uhlig, “Extending the FairRoot framework to

allow for simulation and reconstruction of free streaming data," *J.Phys.Conf.Ser.*, vol. 513, p. 022001, 2014.