Breaking the I/O Barrier: 1.2 Tb/s Ethernet Packet Processing on a GPU

John W. Romein$^{1[0000-0002-1915-5067]}$

ASTRON (Netherlands Institute for Radio Astronomy), Dwingeloo, The Netherlands
romein@astron.nl https://www.astron.nl/

Abstract. Radio telescopes produce enormous amounts of data. Many of them use GPU clusters to combine the digitized antenna signals, usually in real time. Achieving high data rates is challenging: the PCIe bandwidth of discrete GPUs is limited, and without RDMA, handling 200 or 400 Gb/s Ethernet packets with telescope data is difficult.
The NVIDIA Grace Hopper is a novel, innovative system that eliminates the I/O bottleneck of traditional, discrete GPUs by using NVLink instead of PCIe. This opens the door to higher data rates, but faster hardware alone is not enough. In this paper, we combine hardware and software innovations to process Ethernet packets at no less than 1.2 Tb/s, a huge improvement over what was previously possible. We use the Data Plane Development Kit to minimize the receive overhead, and use a new feature that allows packet processing directly by the GPU. We demonstrate the data handling in a correlator application, analyze the performance, and show how to reduce the energy use.
The presented innovations enable the use of GPUs for more powerful telescopes with much higher data rates. The results are also of interest to (GPU) applications from other application domains with high I/O demands, especially if RDMA is not available.

Keywords: GPU · Ethernet · DPDK · Grace Hopper · Radio Astronomy

1 Introduction

Radio telescopes produce enormous amounts of data, and with the relentless drive to build more powerful instruments, these data rates increase and increase. Most telescopes combine the data from tens or hundreds of receivers, to obtain higher sensitivity and image resolution. This data is transported, typically over Ethernet, to a central location where the data is combined by what is called a *correlator*. Due to the high data rates, these data are usually correlated in real time. In the past, this was done using custom-built electronics, ASICs, or DSPs, but nowadays this task is performed by FPGAs or GPUs.

The choice to use FPGAs or GPUs for a correlator depends on multiple factors. Generally, FPGAs are considered to be better at I/O, but difficult to program, while GPUs are better at compute, and allow much more flexible and complex processing pipelines. Instruments with moderately high data rates, like

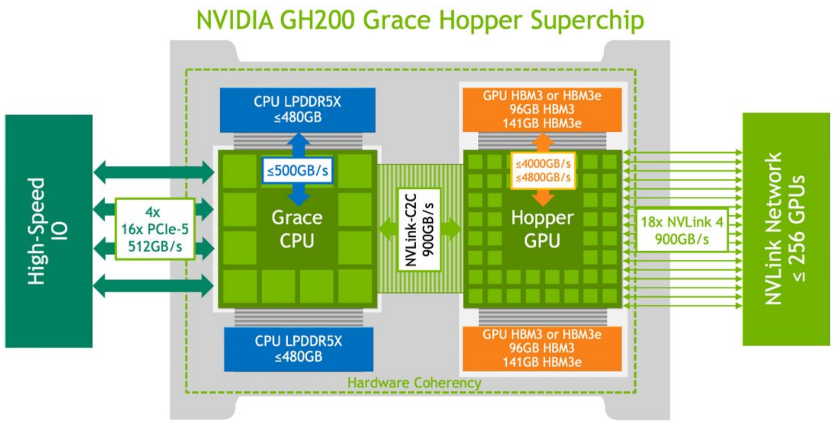


Fig. 1: Schematics of the Grace Hopper Superchip (source: NVIDIA).

CHIME [9], LOFAR [7], and the MWA [13], use GPU-based correlators. However, for instruments with the highest data rates, like ALMA [8] and the two SKA sites, new FPGA-based correlators are being developed, as GPU systems were deemed less suitable to handle tens of terabits per second.

During the past decade, the computational performance of successive GPU generations increased by roughly two orders of magnitude, partly because of the introduction of tensor cores, that can be efficiently used for signal-processing tasks like correlations and beam forming [19, 22]. In the same period, the PCIe bandwidth increased by less than a single order of magnitude, thus the gap between computational performance and I/O performance widened. To profit from the hundreds of tera-ops/s of computational processing power from present-day GPUs, we would need to stream in data at more than a terabit per second — far beyond the 200 or 400 Gb/s PCIe gen 4 or gen 5 bus speeds of recent, discrete GPUs and network interfaces.

The recently introduced *Grace Hopper Superchip* [17] turns out to be a game changer. These innovative systems do not only contain the most powerful GPU to date, the traditional PCIe link between CPU and GPU is replaced by NVLink, that provides seven times more bandwidth than PCIe gen 5 (see Figure 1). This essentially eliminates the I/O bottleneck of discrete GPUs. The figure also shows four PCIe links (on the left), but in practice, Grace Hopper systems have at most three PCIe slots available for network interfaces (NICs). Each slot can hold one 400 Gb/s Ethernet (GbE) NIC or a dual-port 200 GbE NIC, for a total of 1200 Gb/s of Ethernet connectivity, six times more than a PCIe gen 4 GPU can handle.

However, faster hardware alone is not enough to achieve 1.2 Tb/s at the application level. Such data rates cannot be handled by the Operating System (OS): the interrupt, context switching, and packet-copying overheads are prohibitive. And as we will show below, even the CPU memory is too slow to act as a packet buffer. We need techniques that bypass the OS, and stream packet data directly from the NICs into GPU memory. Normally, one would use RDMA techniques like RoCE or GPUdirect [15] for this, but as the data comes from FPGAs, RDMA would severely complicate the FPGA firmware. Instead, we use the Data Plane Development Kit (DPDK) [4] to receive and handle network packets without OS overhead, and we use a recent DPDK addition, called *GPUdev* [5], that allows Ethernet packets to be processed directly by the GPU. This proved to be an essential technique to achieve high data rates.

The main contributions of the paper are the following. Through a combination of software and hardware innovations, we demonstrate a GPU correlator that receives and processes 1196 Gb/s of Ethernet packets, a 6–20-fold improvement over previous-generation GPU correlators. This result shows that in general I/O is no longer the GPU's Achilles heel. We explain the techniques and optimizations that are necessary to achieve this data rate. We analyze the network, CPU, and GPU performance, show how to reduce the energy use, and discuss the strengths and weaknesses of the DPDK approach.

Although this study is driven by the challenges from radio astronomy, the results apply to (GPU) applications from any domain that demands high data rates, especially in situations where the use of RDMA is not possible.

This paper is structured as follows. In Section 2, we provide some background information on radio telescopes, GPU correlators, and DPDK. Section 3 describes several DPDK-based implementations of a GPU correlator, for which we analyze the performance and energy efficiency in Section 4. Section 5 discusses advantages and disadvantages of the DPDK approach, and Section 6 describes related and future work. Section 7 concludes.

The software developed for this publication is available online [1, 2].

2 Background

In this section, we briefly describe how data flows in a radio telescope system, up to the point that telescope data has been combined by what is called the correlator. A complex post-correlator processing pipeline then takes care of calibration and imaging, but this is outside the scope of this paper. Figure 2 depicts the data flow between the antennas and the correlator. On the left, antenna signals are digitized by Analog-to-Digital Converters (ADCs), controlled by FPGAs. The FPGAs also filter and packetize the data. The filter separates the signals into disjoint frequency bands, that can each be processed independently by the different GPU correlator machines on the right. As discussed in the introduction, correlators can be built from either FPGA or GPUs; in this paper we assume a GPU-based correlator. In contrast, the digitizers on the left are always FPGAs, as GPUs cannot control and read ADCs.

As each digitizer FPGA holds the signals from all frequency bands of one antenna, while a GPU correlator node needs one frequency band of all antennas, the data transport from the FPGAs to the GPU systems forms a left-to-right any-to-any pattern, which is known as the "corner turn". In other words: each

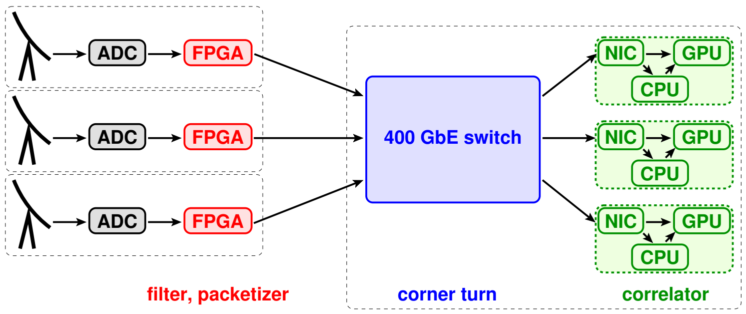
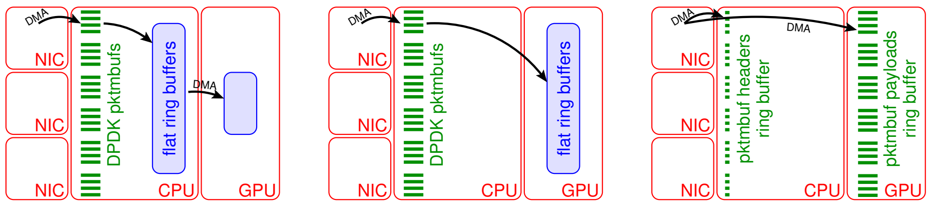


Fig. 2: Data flow from antennas to GPU correlator systems.

FPGA sends packets to all GPU correlator systems, and each GPU correlator system receives packets from all FPGAs. The corner turn is performed on the network switch in the middle of the figure, which is physically close to the GPU correlator systems.

Depending on the telescope, the FPGAs digitizers and the GPU correlator systems may be any distance between a few meters and thousands of kilometers apart (the distance between the two outermost antennas is one of the factors that determines the eventual image resolution). We use Ethernet as the transportation method, because Ethernet is well supported by both the FPGAs and by the GPU systems. Moreover, Ethernet works over any distance. Often, these Ethernet packets are formatted as UDP/IP packets, so that they can be routed. By design, the data transport is unreliable, as a reliable protocol like TCP would severely complicate the FPGA firmware and requires additional buffering, while in a real-time environment there is generally no time for retransmissions anyway. Also, the post-correlator processing pipeline may discard data for other reasons (in particular, due to Radio Frequency Interference), so the pipeline is well capable of handling missing data. Apart from UDP/IP/Ethernet headers, the packets contain an application-specific header with a timestamp of the precise time that the first sample in the packet was taken, the antenna number, and the frequency band number. The packet payload typically contains a few thousand consecutive (filtered) antenna samples, so that the size of the whole packet does not exceed the jumbo frame limit (9000 bytes), while the header size is small compared to the payload size.

A complicating factor of a correlator application is that we cannot assume that the data from all antennas arrive at the same time in a correlator system. Packets from a thousand kilometer distant antenna may arrive tens or even a hundred milliseconds later than from a nearby antenna. To combine the samples, the input streams from the different antennas must be realigned. Therefore, each correlator system maintains a ring buffer in which the input data is buffered, typically for hundreds of milliseconds or a few seconds. For each antenna, a separate ring buffer is used. The ring buffer is selected by the antenna-number value



(a) 2-copy (b) 1-copy (c) 0-copy

Fig. 3: Copy behavior of the different implementations.

in the packet header, and the location within the ring buffer where samples from the packet payload are stored is determined by the header's timestamp value modulo the ring-buffer size. The ring-buffer data is continuously overwritten by newer data, so there is only a limited amount of time available to process the data. The data also needs to be available in the buffer while it is being transferred to the GPU.

This paper focuses on the receipt and processing of the network packets in the GPU systems, on the right in Figure 2. Whereas in the 10 and 40 GbE era UDP/IP packets could be received through the operating system, the interrupt, context switching, and packet copying overheads are too large for 100 Gb/s Ethernet and beyond. Hence, we need a mechanism that bypasses the operating system in the critical receive path. As we strive to achieve line speeds from *multiple* network interfaces, we choose the Data Plane Development Kit (DPDK), also because of its recently added support for GPUs.

DPDK is a toolkit that allows an application to take full control over a network interface, bypassing the OS. The main functions are the functions that take care of receiving and sending Ethernet packets, and the ones that manage packet buffers in memory pools. Any network protocol on top of Ethernet also has to be implemented in user space. For simple protocols like UDP/IP, the application will likely implement the protocol stack itself, but for complex protocols like TCP/IP, it is probably more convenient to use an open-source user-level library like F-Stack [3]. As a DPDK application can send and receive any Ethernet packet on a network, which is a severe security threat, DPDK applications run with an elevated privilege level (as superuser, or with some specific capabilities). The toolkit is supported by all major NIC vendors, but the GPUdev extension currently only works with NVIDIA NICs and NVIDIA GPUs. We elaborate on the use of DPDK in the next section.

3 Implementation

We implemented three correlator variants. Each of them uses DPDK, but the data paths are different. Figure 3a depicts how the most straightforward implementation of a GPU correlator works: the packets are received in DPDK packet buffers, the packet data is copied into the ring buffers in CPU memory, and after some time, when all data for a particular time interval has arrived (or should have arrived), the section of ring-buffer data for that time interval is copied to GPU memory, for each antenna. Subsequently, the GPU processes the data. We call this the *2-copy* variant, as receiving data from the NIC is generally not counted as a "copy" action.

The second variant does not have the ring buffers in CPU memory, but in GPU memory (see Figure 3b). Here, the packets are still received in DPDK packet buffers, but their contents are copied into the GPU ring buffers. We call this the *1-copy* variant. Alternatively, we could have stored the ring buffers in CPU memory and let the GPU channel filter kernel read the ring buffer contents through unified memory, but despite the use of NVLink, this makes the channel filter kernel prohibitively slow, so we do not further consider this alternative.

The third variant, the *0-copy* variant, works quite differently (see Figure 3c). Rather than implementing the ring buffer as a flat memory buffer, we use a giant amount of DPDK packet buffers as "ring buffer", and process the data at a later moment directly from the packet buffers. For this, we use the recently added GPU support in DPDK, *GPUdev* [5], that allows allocating packet buffers in GPU memory. DPDK also allows splitting packets, and receive packet fragments in different memory pools. We allocate one memory pool in CPU memory and another memory pool in GPU memory, and split incoming packets so that each packet header (56 bytes) is received in CPU memory, and the associated packet payload (8192 bytes) is received in GPU memory. Important to note is that 99.3% of the packet data is directly DMAed from the NIC into GPU memory, bypassing CPU memory. The packet headers contain the timestamps and antenna numbers, that determine the location in the ring buffer. The timestamps and antenna numbers are inspected by the CPU, and a pointer to the packet payload (in GPU memory) is placed in the ring buffer. The ring buffers thus contains pointers to packet payload buffers instead of the samples themselves. After all packets from a certain time interval should have arrived, the CPU copies the ring buffer payload pointers to the GPU, and launches the filter and correlator kernels that we describe in Section 3.1.

Even though we use almost 74 GB (77%) of the GPU memory for buffering packet payloads, the buffer fills up quickly at high data rates. At the maximum data rate of 1.2 Tb/s, the 9.4-million entry packet buffer is completely filled after only 0.52 seconds. The GPU processes blocks of 18 GB of data (one quarter of the ring buffer size), which corresponds to 0.13 seconds at the highest data rates, so that the remainder of the ring buffer is used to receive new packets, while there is also sufficient time to overcome the packet arrival time differences from the different antennas.

At first, using large amounts of GPU memory with GPUdev did not work well. During program initialization, the GPU memory is registered by DPDK to make it accessible to the NICs, but registering a few gigabytes of GPU memory was prohibitively slow (taking hours), and was exponentially slower for even larger sizes. We fixed this in DPDK's mlx5 driver, by sorting a list of segments only once instead of for each added segment. A patch is available [2].

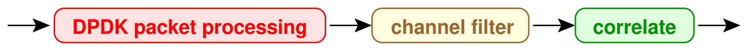


Fig. 4: Simplified view of the GPU processing pipeline.

We distribute the packet-handling work over multiple CPU cores, by using multiple receive queues. Each receive queue is associated with one CPU thread that continuously polls the queue for incoming packets. The amount of receive queues (and thus CPU cores) is an adjustable parameter. The antenna streams are divided over the available receive queues; consecutive packets from the same antenna always end up in the same receive queue.

As all cores and CPU memory in the Grace CPU are in the same NUMA domain, there is no need to bind threads to specific cores. Yet we distinguish between threads that poll a NIC receive queue and threads that do not; the polling threads make as few system calls as possible (to not stop receiving packets for an extended period of time), while (blocking) system calls are performed by the latter group of threads.

DPDK's per-core memory pool cache plays a crucial role in obtaining good performance. This is a 512-entry per-core cache where a core can quickly allocate and deallocate packets from, instead of using the much slower shared pool. We found that, regardless of how many cores are used, data rates beyond 800 Gb/s are impossible without effective use of the memory pool cache. This limits the freedom in application design choices. First, one cannot receive a packet by one core, hand it over another core that manages the GPU, and let the GPU-managing core deallocate the packet when the GPU is ready, because the receiving core would always encounter an empty cache and the deallocating core a full cache. Second, one cannot deallocate large amounts of packets in one go (even though a burst deallocation function is available). Instead, after a packet has been processed, we defer packet deallocation, so that every time a new packet is received, another unused packet is deallocated. This keeps the cache usually in a partially filled state.

3.1 GPU processing

Although a comprehensive study of the GPU kernels is outside the scope of this paper, we briefly describe the signal-processing operations performed by the GPU. The GPU performs three major tasks: it handles the input packets (this only applies to the 0-copy variant), and subsequently channelizes and correlates the signals (see Figure 4). The last two tasks are standard signal-processing operations that basically every (GPU) correlator performs.

Even though packet handling and channel filtering are depicted as two operations, they are performed by the same GPU kernel. The channel filter comes from a newly developed GPU library, that combines a PolyPhase Filter Bank (FIR filters and FFT), delay compensation (optional), bandpass correction (optional), and a memory transpose in a single GPU kernel. All these operations are fused in a single kernel, to reduce the number of GPU memory accesses. The library compiles the GPU code at run time, applying specific optimizations that depend on the given number of antennas, frequency channels, etc. We use the runtime compilation property also to provide the library with custom GPU code that reads input data from DPDK packet payloads, rather than a flat memory buffer. This is a nice separation of responsibilities: the filter library itself is oblivious of DPDK, yet the GPU channel filter kernel is heavily involved in processing the DPDK packets.

The second GPU kernel is a correlator kernel that combines the antenna data by pairwise multiplication and integration of the filtered antenna samples. This kernel comes from the *Tensor-Core Correlator* library [22], that performs these operations on tensor cores. It is the fastest (GPU) correlator to date.

Once the data from all antennas should have arrived in the ring buffers (possibly with a time offset), the GPU processes blocks of 18 GB of data; a quarter of the ring buffer size. Consecutive blocks are slightly overlapping, as the FIR filters in the channelizer are in fact convolutions that need some historical data from the previous block. This complicates the implementation. Due to the large data blocks on which the GPU operates, the kernel launch overhead is negligible. Yet, the kernels are launched by a different thread than the CPU threads that receive DPDK packets, as the kernel launches may involve (blocking) system calls.

We considered using a persistent GPU kernel that would poll for new incoming packets, and immediately process (channelize) new packet data. The advantage of that would be that the packet-buffer size could be kept small, decreasing the DPDK overhead. However, it may increase the energy use, as it keeps the GPU busy at all times, even if there is no new data to process. Unfortunately, a persistent kernel is difficult to implement in this case, for several reasons. In particular, the realignment in time of the channelized data prior to correlation (which requires some sort of ring buffer as well), but also the missing support for persistent kernels in the filter library, the internal state that a filter must maintain, the deallocation of processed packets, and CPU–GPU synchronization all add to the complexity. Therefore, we refrained from using a persistent kernel.

3.2 Correlator output

In this study, we do not specifically optimize for high output data rates, as the output data rate of a correlator is typically (much) lower than the input data rate. If the output is written to file, we use the new cuFile library from the CUDA toolkit to write data directly from GPU memory to file. Alternatively, the data can be sent over a TCP connection to an external system. For simplicity, we do this via the operating system (and another virtual instance of one of the NICs), which works fine for speeds up to about 50 Gb/s. If the output data rate requirements for a specific instrument setup would exceed this, the output data path should also be optimized (either through DPDK, or some RDMA protocol), but at such high output data rates, the biggest challenges would not be in the correlator itself, but in the post-correlator processing pipeline.

4 Performance

We analyze the application and DPDK performance on the CPU and GPU. Although in reality antenna data are digitized and packetized by FPGAs near the individual antennas, in this experimental setup we use a CPU-based packet generator that mimics the FPGA behavior, for practical reasons. We only evaluate the performance of a single GPU correlator system, as multi-GPU correlator systems operate independently of each other, since they each process a different frequency band, so multi-GPU scaling is trivial. The corner turn does *not* scale trivially, and can impose high packet-switching requirements on the switch, but this is outside the scope of this paper.

The measurements were performed on two QCT S74G-2U Grace Hopper systems with 96 GB HBM3 and 480 GB LPDDR5 memory (one for the packet generator, the other one for the GPU correlator), using a patched version of DPDK 24.03 (see Section 3) and Linux kernel 6.5.0-1024-nvidia-64k. For availability reasons, we use a mixture of 400 GbE and dual-port 200 GbE ConnectX-7 NICs, for a total of 1200 Gb/s per system (in each direction). To simplify the software, we split the 400 GbE NICs into two virtual 200 GbE NICs, so that the software does not need to distinguish between different link speeds. In practice, we saw that a single 400 GbE link and two bundled 200 GbE links behaved similarly. SSH connections are routed via a separate USB-Ethernet dongle, to not interfere with the high-speed network interfaces. Unless stated otherwise, 12 (out of 72 available) CPU cores are used to poll the NICs and insert the packets in the circular buffer.

We simulate 72 antennas and 8 bits per sample, typical numbers for a radio telescope. For other instruments, the ratio between the amount of I/O and computations may be different. The I/O bandwidth scales proportionally to the number of antennas, but the amount of computations scales quadratically. Fewer bits per sample reduces the amount of I/O, but has no impact on the amount of computations.

We seek for the largest amount of data that a Grace Hopper system could receive and process in real time, without packet loss (the correlator should normally not lose data, even though the remainder of the processing pipeline tolerates it). Figure 5 shows the obtained network bandwidth, for each of the three variants. The 2-copy variant runs up to 309 Gb/s without packet loss. The 1-copy variant works best with 24 cores, and achieves a data rate of 670 Gb/s. The 2-copy and 1-copy variants can actually process higher data rates, but start to drop packets then, something that we wish to avoid. Only the 0-copy variant is able to process packets at 1.2 Tb/s. This means that the application is still limited by network bandwidth, but this bandwidth is six times higher than what was possible with previous-generation GPUs. And as we will see later, the GPU would not be able to handle a much higher data rate, so the GPU compute power and I/O capabilities are fairly balanced.

Figure 6 shows the actual CPU memory bandwidth use, which is a scarce resource for the 2-copy and 1-copy variants. Unfortunately, there are no performance counters that measure the CPU memory bandwidth use directly, so we

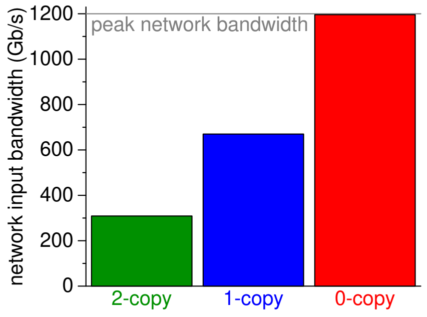


Fig. 5: Achieved input network bandwidth (without packet loss).

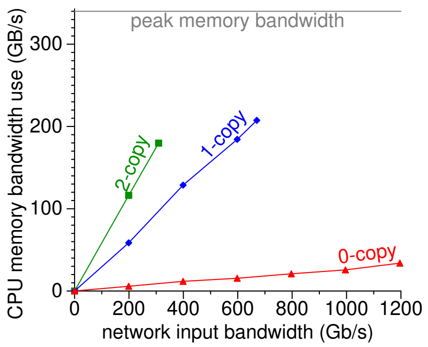


Fig. 6: CPU memory bandwidth use.

measured and added the memory traffic from the local CPU cores, NVLink, and PCIe busses, as described in the *Grace Performance Tuning Guide* [18].

From the figure, we see that the 2-copy and 1-copy variants would never scale to 1.2 Tb/s, as their extrapolated graphs exceed the memory bandwidth that is available. A separate benchmark measured the memory bandwidth at 340 GB/s, which is consistent with what the Tuning Guide [18] reports. Yet, we see that the 2-copy and 1-copy variants do not get close to the 340 GB/s bandwidth use as they start losing packets; this already happens around 200 GB/s. For the 2-copy variant, this is mostly due to the irregular CPU-to-GPU transfers: due to the high NVLink link speed, these transfers can claim all the available CPU memory bandwidth, leaving insufficient memory bandwidth for packet receipt. The 1-copy variant does not suffer from this, still the memory access pattern is not optimal to reach full bandwidth without packet loss.

In fact, the 0-copy variant is the only variant that does not suffer from insufficient CPU memory bandwidth; the figure shows that its actual memory bandwidth use is low. In the remainder of this section, we only consider the 0-copy variant.

Note that, unlike CPU memory bandwidth, GPU memory bandwidth is amply available; the 150 GB/s that is needed to store packet payloads, is only a small fraction of the 3.5 TB/s that is available.

4.1 Energy efficiency

A well-known technique to improve the energy efficiency of a GPU (and many other processors) is to run the application at a reduced clock frequency [21]. As the GPU idles for 23% of the time at the default (and highest supported) clock frequency, there is room to reduce the clock speed, down to the point that it just does not lose real-time behavior (see Figure 7). This way, we can reduce the energy consumption already by 96 Watts.

Similarly, we can reduce the CPU clock frequency (see Figure 8). Note that, unlike the GPU kernels, the packet-handling CPU cores never idle but keep on

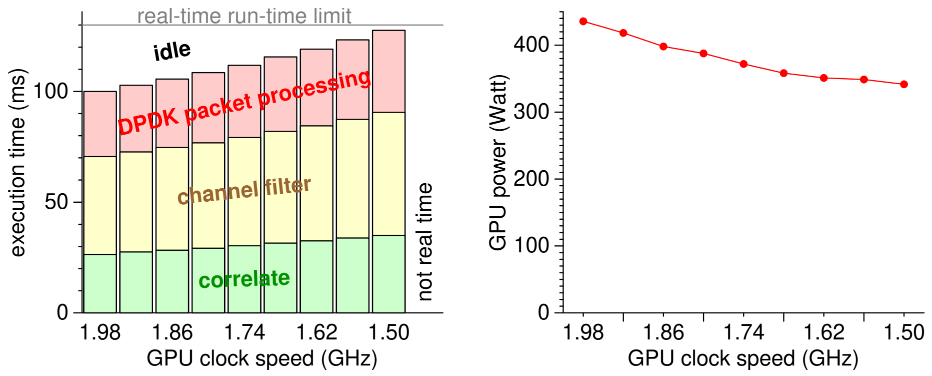


Fig. 7: GPU performance and energy efficiency. On the left: the time spent in the different kernels, as a function of clock frequency (note that the DPDK packet handling is in fact part of the filter kernel, but its execution time is shown separately). On the right: the power use.

polling the NICs to check if new packets have arrived. When using only 6 CPU cores to handle the incoming packets, there is hardly any room to reduce the clock frequency without packet loss. However, if we increase the number of polling cores to 12, we can decrease the clock frequency all the way down to 1.8 GHz, without observing packet loss. This way, we can decrease the CPU power from 119 Watts (with 6 fast-running cores) to 69 Watts (with 12 slow-running cores). We see no additional benefit from increasing the number of polling cores to 18, because the clock speed cannot be reduced any further without packet loss.

5 Discussion

So far, we learned that the DPDK approach, and in particular its recent support to receive packet payloads in GPU memory, yield extremely high data rates on streaming data, and good application performance. However, there are some drawbacks to this approach. First, the DPDK model provides no method to control *where* packet payloads are received in GPU memory. As a result, the GPU spends 22%–27% of the time collecting the input data from the scattered packet payloads: looking up an input sample requires an extra pointer indirection, and GPUs have no huge page support to reduce the amount of TLB misses.

Second, splitting packets increases DPDK's internal CPU overhead, because the header and payload buffers are allocated and deallocated from separately managed memory pools. We tried receiving the full packet (header and payload) in GPU memory. As, in this case, the relevant metadata (timestamp, antenna number) are in GPU memory, either the CPU needs to retrieve this data from GPU memory so that it can place the packet in the circular buffer, or the GPU itself should fully handle the packet and maintain the circular buffer itself. Unfortunately, for the former approach, the DPDK toolkit refused to register unified

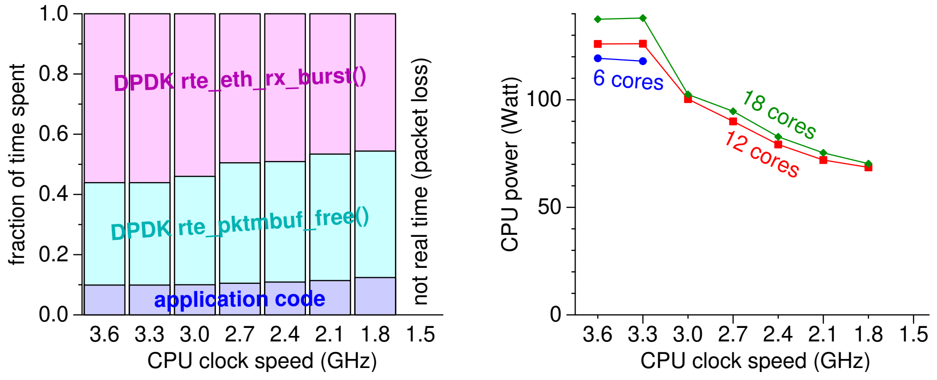


Fig. 8: CPU performance and energy efficiency. On the left: the time spent in DPDK and in the application code, as a function of clock frequency. On the right: the energy use.

(managed) memory, so that the memory would have been accessible by the NIC, CPU, and GPU. And the latter approach is difficult to implement. For example, when the GPU finished processing a packet, it cannot return the packet fragments to the mbuf pool and should leave this to the CPU, but then it is difficult to make efficient use of DPDK's mbuf cache, which is crucial for performance.

Finally, the presented 0-copy method is difficult to integrate into existing GPU correlator applications like the LOFAR [7] and AARTFAAC [20] correlators, due to the different way in which input data is stored, the different roles of CPU threads, and the prohibitive costs of thread synchronization for every received packet. The application that we use for this demonstration, was written from scratch (except for the GPU kernels) so that DPDK could be properly integrated.

On the other hand, we have not seen any competing approach that yields such high data rates, and given the enormous increase in achieved data rates, we consider the DPDK and GPUdev solution as a major step forward.

6 Related and future work

So far, GPU correlators have only been built for radio telescopes with moderate data rates, like CHIME (for a total of 6.6 Tb/s divided over 1024 GPUs) [9], LOFAR (236 Gb/s) [7], AARTFAAC (120 Gb/s) [20], and MWAX (11 Gb/s) [13], to name a few. It is also worth noting what has *not* been built: so far, GPU and NIC technology were not ready for instruments like ALMA, which is currently being upgraded to 63 Tb/s [8]. This study shows that a 63 Tb/s GPU correlator is feasible. But we also need more efficient packet handling in the AARTFAAC and LOFAR correlators, as their networks are currently being upgraded to 400/100 GbE, allowing much higher data rates.

Others tried ibverbs to bypass the operating system, e.g., in the SPEAD2 packet-handling library [12] or in an experimental setup [10]. The performance that was obtained with it in practice, fell 10–30% short of the used Ethernet line rates (200 and 400 Gb/s, respectively).

NVIDIA Holoscan [16] is a framework that supports implementing applications for real-time sensor processing on GPUs. Holoscan is used in a setup with the Allen Telescope Array, where Holoscan's Advanced Network Operator is used to process 100 GbE data, bypassing the CPU [11]. As Holoscan is built on top of DPDK, it should be able to perform similarly to what we report.

The DOCA toolkit [14] is a collection of SDKs that provide (low-level) access to NVIDIA NICs and DPUs, some of which allowing packet receipt in GPU memory (e.g., DOCA Ethernet and DOCA GPUnetIO). A recent addition to the toolkit, DOCA DPA (and the driver layer DOCA FlexIO) is highly promising, as it allows programming the Data-Path Accelerator with application-specific code. As a next step, we plan to explore DOCA DPA, and try to run code on the NIC that inspects the header of an incoming packet, computes a destination address within a flat ring buffer in GPU memory (based on the header's antenna number, frequency band number, and timestamp), and DMAs the packet payload directly from the NIC into the GPU ring buffer. If we can make this work, it would nearly eliminate the CPU and GPU overhead that we encounter with DPDK.

7 Conclusions

Many radio telescopes use GPU clusters to process antenna data, in real time. The desire to build more powerful radio telescopes results in higher data rates, that must be handled by such GPU systems. However, the I/O bandwidth of successive, discrete GPU generations did not keep pace with the increase in computing power, and, in the absence of RDMA, truly efficient methods to receive and handle network packets with telescope data were missing.

This paper demonstrates an enormous increase in network packet processing rates, through a combination of hardware and software innovations. The Grace Hopper architecture eliminates the PCIe bandwidth limitation, the Data Plane Development Kit (DPDK) removed the operating system overhead of receiving network packets, DPDK's recent support for GPUs allowed receiving packet data directly from the network interface into GPU memory, and the GPU application processes network packet payloads. Without these innovations, handling packets at a few hundred Gb/s was already difficult, but we demonstrate that 1.2 Tb/s per GPU is possible now, with a reasonable balance between GPU compute performance and I/O capabilities. The performance analysis shows that the CPU and GPU overheads from handling the network packets is noticeable but not prohibitive. We also showed that tuning clock frequencies led to a 145 Watts reduction in energy use, without losing real-time performance. Future work targets methods that further reduce the overhead, but the presented approach already enables the processing of much higher telescope data rates.

Acknowledgments. This work received funding from the European Southern Observatory through the ALMA GPU correlator study; from the European Commission through the RADIOBLOCKS grant (HORIZON-INFRA-2022-TECH-01, Grant Agreement 101093934); from the Netherlands eScience Center/SURF through the RECRUIT and PADRE grants; and from the Netherlands Organization for Scientific Research (NWO) through the DAS-6 [6] grant. We thank Cliff Burdick and Adam Thompson from NVIDIA for their advise.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. DPDK correlator, https://gitlab.eso.org/alma/gpu-correlator/dpdk-correlator
2. DPDK mlx5 patch, https://www.astron.nl/~romein/dpdk-mlx5-24.11.patch
3. F-Stack, https://www.f-stack.org/
4. The Data Plane Development Kit, https://www.dpdk.org/
5. Agostini, E.: Boosting Inline Packet Processing Using DPDK and GPUdev with GPUs (April 2022), https://developer.nvidia.com/blog/optimizing-inline-packet-processing-using-dpdk-and-gpudev-with-gpus/
6. Bal, H., et al.: A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer **49**(5), 54–63 (May 2016)
7. Broekema, P.C., et al.: Cobalt: A GPU-based correlator and beamformer for LOFAR. Astronomy and Computing **23** (April 2018)
8. Carpenter, J., Brogan, C., Iono, D., Mroczkowski, T.: The ALMA2030 Wideband Sensitivity Upgrade (2022), https://arxiv.org/abs/2211.00195
9. Denman, N., et al.: A GPU Spatial Processing System for CHIME. Journal of Astronomical Instrumentation **9** (September 2020)
10. Liu, W., Burnett, M.C., Werthimer, D., Kocz, J.: A 400 Gbit Ethernet Core Enabling High Data Rate Streaming from FPGAs to Servers and GPUs in Radio Astronomy. Astronomical Society of the Pacific **136**(12) (December 2024)
11. Ma, P.X., et al.: A Deployed Real-Time End-to-End Deep Learning Algorithm for Fast Radio Burst Detection. under review (2025)
12. Merry, B.: SPEAD2, https://spead2.readthedocs.io/
13. Morrison, I.S., et al.: MWAX: A new correlator for the Murchison Widefield Array. Publications of the Astronomical Society of Australia **40**, e019 (2023)
14. NVIDIA: DOCA, https://docs.nvidia.com/doca/sdk/doca+ethernet/
15. NVIDIA: GPU Direct, https://developer.nvidia.com/gpudirect/
16. NVIDIA: Holoscan, https://www.nvidia.com/en-us/clara/holoscan/
17. NVIDIA: GH200 Grace Hopper Superchip Architecture (2024), https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper/
18. NVIDIA: Grace Performance Tuning Guide (2024), https://docs.nvidia.com/grace-perf-tuning-guide/measuring-performance.html
19. Oostrum, L., et al.: The Tensor-Core Beamformer: A High-Speed Signal-Processing Library for Multidisciplinary Use. In: IEEE IPDPS'25. Milan, Italy (June 2025)
20. Prasad, P., et al.: The AARTFAAC All-Sky Monitor: System Design and Implementation. Journal of Astronomical Instrumentation **5**(4) (December 2016)
21. Price, D., et al.: Optimizing performance-per-watt on GPUs in high performance computing. Computer Science – Research and Development pp. 1–9 (2015)
22. Romein, J.W.: The Tensor-Core Correlator. Astronomy and Astrophysics **656(A52)**, 1–4 (December 2021)