Efficient Data Pipelines for Combined FPGA/APU Systems

Oskar Flordal, Unibap AB

Problem statement

A common architecture for COTS payload computers in space applications combines a COTS APU (CPU+GPU combo) with an FPGA. The FPGA provides monitoring, Triple Modular Redundancy (TMR), and other space-suitable properties, while the APU serves as the high-performance compute component. The FPGA can also be utilized for custom interfaces and tasks it excels at, such as certain real-time operations.

However, this architecture presents several challenges that need to be addressed to optimize the APU's usage in a space context:

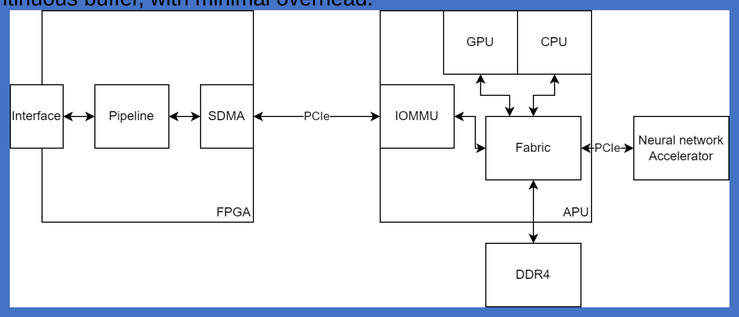
- PCIe Transfer Latency:
  - The relatively long latency of PCIe transfers between FPGA and APU can impact system performance.
- Real-Time Handling on Non-Real-Time Systems:
  - Managing functions like Pulse Per Second (PPS) on the APU, which is typically not a real-time system.
- Inter-Component Communication:
  - Handling interrupts and communication between the APU and FPGA without compromising performance.
- Memory System Optimization:
  - APUs have highly optimized memory systems that may not align with traditional FPGA IP designs for space applications. For example:
    - Requirement for 64/128 byte bursts
    - Restrictions on byte-granularity writes in LPDDR5, which hasn't been standard practice in FPGA implementations
- Space-Specific Interface Integration:
  - Integrating space-specific interfaces within the Linux operating system environment.

Addressing these challenges is crucial for effective utilization of COTS payload computers in space applications, ensuring optimal performance and reliability in the unique constraints of space environments.

Solution on the FPGA

To feed a pipeline like the SAR focus pipeline shown on the right, you would typically connect an imager via an LVDS or high-speed serial interface. These are not usually available as COTS components suitable for space use, so we rely on FPGA implementation to bridge this gap.

For increased efficiency and minimal overhead, a design is built with properties similar to user-space allocated buffers in V4L2. The user can allocate a set of buffers using hipHostMalloc that can be accessed by both the CPU and GPU without additional copies. These are then passed to the driver, which maps them into continuous space as seen by the FPGA via the system's IOMMU. The FPGA implementation can then write directly into GPU-owned memory as if writing to a continuous buffer, with minimal overhead.



When routing data from the backend interface to the APU, the FPGA can perform preprocessing to simplify the data at no additional cost as it streams by. For SAR data, this can include subsampling or even pre-tiling the data to optimize memory access patterns later.

Using 10 Gigabit Ethernet

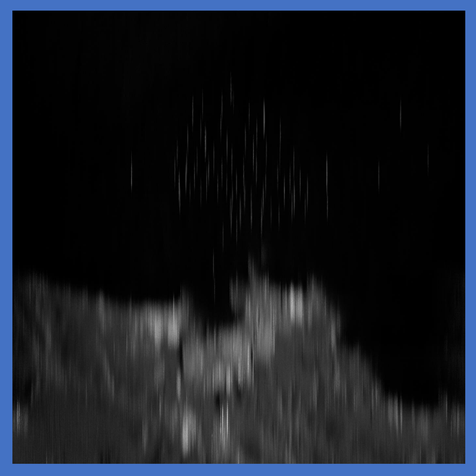
On modern satellites, the interface to the imager could be 10 Gigabit Ethernet. On the iX10, this is connected directly to the APU. However, receiving data at 10 Gbit/s speeds is challenging on a regular system, especially for UDP workloads, where packet loss is almost guaranteed even at lower speeds.

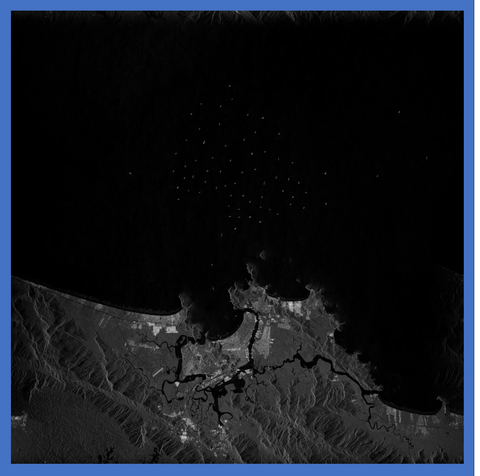
To address this effectively, it's advisable to use low-level libraries that bypass the Linux network stack. For example, DPDK has a driver for the AMD 10 Gigabit MAC that allows direct access to the DMA. This presents a future optimization opportunity.

Example application SAR focus

SAR focusing, converting L0 I/Q SAR data to Single Look Complex, has traditionally been challenging with conventional space solutions like FPGAs but is well-suited to GPUs in modern COTS computers like the Unibap iX10. While there are various SAR focusing methods, fast algorithms typically involve applying FFTs, inverse FFTs, and element-wise multiplication of matrices (or matrix/vector).

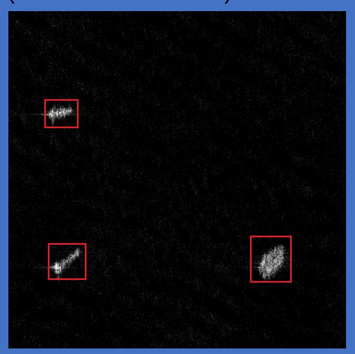
In this case, we implement the simplest Range Doppler Algorithm based on an open-source implementation, using Sentinel-1 Stripmap L0 data sourced from Copernicus.





To optimize the implementation, we design the focusing algorithm as a multi-stage pipeline where each step passes buffer references to an asynchronous FIFO:

1. Sourcing the data from a network interface or local storage (for demo purposes)
2. Focusing step running as back-to-back kernels on the GPU
3. Running a neural network accelerator to detect ships in the image using an off-the-shelf object detector (YOLOv8s in this case)

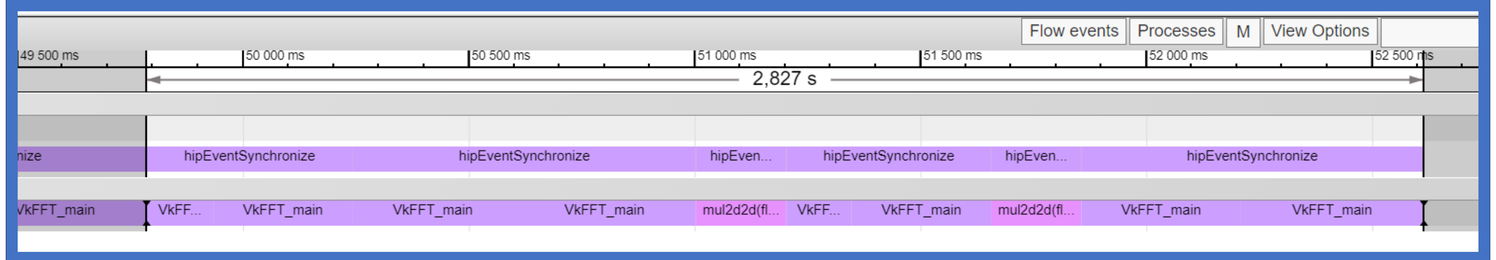


The application is written in ROCm/HIP to utilize the AMD GPU for FFTs and matrix multiplications. We use VkFFT with the HIP backend for FFTs, processing 2^14 * 2^14 sample tiles (~268 Msamples).

This implementation is severely memory bandwidth limited with one pass consuming 47 GB of read/write on a memory system that can do 36 GB/s. Individual jobs peak at 32 GB/s.

To enhance performance, we apply the following optimizations:

- Pre-calculate filters for combined range and RCMC, as well as azimuth
- Crop data to an even power of two (16384x16384 samples) for optimal FFTs and memory access patterns while minimizing seams between tiles
- Merge range and RCMC filters
- Allocate all buffers as host-allocated pointers, avoiding copies between CPU and GPU in this shared memory system



This implementation is severely memory bandwidth limited, with one pass consuming 47 GB of read/write on a memory system capable of 36 GB/s. Individual jobs peak at 32 GB/s.

The SAR focusing loop alone takes 2.8s, achieving a throughput of 95 Mpixel/s. Being mostly memory bandwidth bound, several opportunities for improvement exist:

- Read data in integer format for the first loop
- Fuse the FFTs around the filters
- Write directly to an 8-bit format for analysis
- Build the azimuth filter in the filter kernel instead of reading it from disk. These optimizations could potentially increase speed to 2s per tile or 135 Mpixel/s.

System aspects

When running the filter with an accelerator, additional fabric/PCIe bandwidth is consumed by pushing the 8-bit image to the accelerator. The accelerator can run the object detector at ~160 Mpixel/s, incurring a bandwidth penalty of ~2 Gbit/s when sending data as NV12 (for convenience). Sourcing the data requires at least an additional 3.2 Gbit/s at 135 Mpixel/s for 2*12bit per sample.

Key insights for efficient system-level operation:

- Efficient use of memory bandwidth is crucial
- When copying data from the FPGA from an LVDS or HSS-based interface, it's important to copy the buffer directly into user memory, specifically into a user GPU-visible buffer
- Given that the APU balances power use between CPU/GPU and memory system, it's advisable to keep CPU usage low and utilize efficient DMA to handle the workload

UNIBAP