Real-Time Edge Processing During Data Acquisition

Max Rietmann(✉), Praveen Nakshatrala, Jonathan Lefman, and Geetika Gupta

NVIDIA Corporation, Zürich, Switzerland
mrietmann@nvidia.com
https://www.nvidia.com

Abstract. The next generation of high-intensity light sources, microscopes, and particle accelerators enable exciting new insights and discoveries. However, the data rates generated by these sophisticated instruments are exploding due to higher sensor scan rates and increased resolution. In parallel, the vision connecting experiments with real time feedback, steering, and integration demands new solutions in both hardware and software. An edge-supercomputer co-located with the sensors or instruments combined with a larger supercomputer enables real-time processing of streaming experimental data at the edge with resource intensive analysis, simulation, and reconstruction at the larger cluster.

Today, post-acquisition data processing is expensive in terms of time as well as storage, and it is scientifically costly since many opportunities are missed during data acquisition. We will describe how a small computational infrastructure can reduce the cost and latency to using the data as it is generated.

Using applications in ptychography and light sheet microscopy as examples, this paper will show how to build data streaming pipelines that form the foundation for real-time processing, visualization, feedback, and steering. We will show how a developer can write high-performance data processing pipelines using Python and C/C++ to integrate traditional processing with the latest ML and AI techniques. We highlight end-to-end performance profiling and optimization as well as the libraries and frameworks from NVIDIA to build these application-driven processing pipelines from edge to computing center.

This work pushes us towards the vision of realizing an end-to-end workflow starting with streaming directly from the instrument at the edge to the data center.

Keywords: HPC@Edge · GPU computing · Streaming processing · Visualization

1 Introduction

The next generation of high-intensity light sources and advanced microscopes will lead to exciting new scientific discoveries and insights. With enhanced resolution and increased speeds of sensor-data acquisition, these sophisticated instruments produce data at high volumes and velocities. To manage this data deluge, streaming processing pipelines are a key building block to ensuring timely results, live visualization, and feedback control. Such real-time processing of data pipelines from the edge to the computing center warrants novel software and hardware enhancements.

Often, various processing steps are involved in the analysis of data and typically, a data-processing pipeline is employed. Staging data through non-volatile I/O between these processing steps, though common, is a bottleneck to achieving real-time analysis of data at scale. To address these concerns, this paper presents a suite of solutions for developing streaming reactive pipelines, GPU-accelerated libraries for data ingestion and processing.

Despite the accelerating computational requirements, most scientists have embraced Python-based open-source frameworks because of the ease of prototyping and iterating on new implementations. Computational performance optimization is often pushed to the final stages of deployment, after a satisfactory implementation is in place. Frameworks and libraries like NumPy and PyTorch are often employed. This paper will address improving computational throughput and efficiency for real-time experiments without dramatic changes to current code bases.

This paper will cover topics including:

– Commonalities across these sensor-driven processing pipelines, particularly the transition from high-data rate processing (frame-by-frame) to high-corpus processing (collective processing and analysis of many frames).
– Example applications include x-ray ptychography and lattice light sheet microscopy; covering common workflows, data rates and processing requirements.
– Performance optimization results obtained through the adoption of GPU-accelerated frameworks like JAX and CuPy.
– Introduction of a new streaming data-processing library, NVIDIA Streaming Reactive Framework (SRF). This library enables developers to build high-performance data-pipelines containing several processing stages.
– Conclusions and discussion on what can be realized today and the future for streaming processing pipelines.

2 Common Sensor-Driven Workflows

Typical real-time sensor-driven experimental processing pipelines have common patterns. Considering scientific and manufacturing instruments, sensors generate raw data streams which are consumed and processed into downstream data components. In Fig. 1, raw data from sensors are corrected and normalized, forming sub-components of data. An example of such a process is dark current compensation for electron-counting cameras. These sub-components are further assembled into minimal data units. An example of assembling data sub-components into a minimal data unit is correcting motion from electron beam induced motion from a cryo-EM datasets [15]. Finally, minimal data units assembled or reconstructed into a complete finalized dataset. An example of creating a finalized dataset is aligning and mutually reconstructing multiple overlapping x-ray tomographic projections [5].

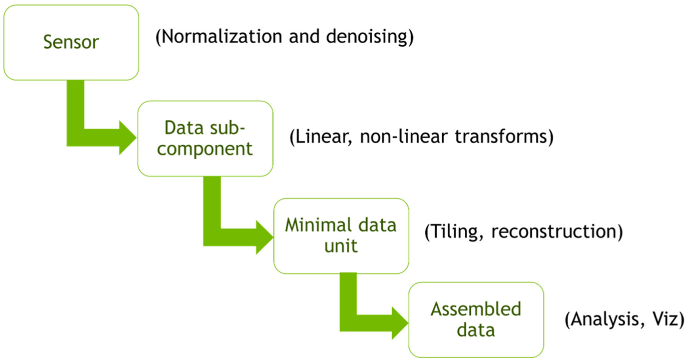

Fig. 1. Typical processing and data flows across real-time instrument processing pipelines

This generic workflow model applies to many use cases, and a critical property of these data-processing pipelines is the change of the data rate through the application and how dependencies across data units change through the pipeline.

2.1 Data Rates and Data Burden

Many applications tend to stage data through non-volatile I/O because real-time processing has not been achievable, thus data streams are stored. Using files from storage is a limitation to make real-time processing schemes; reading data from disk takes as much or more time than processing. To meet high data rates and achieve real-time data processing, it is imperative to move toward an in-memory streaming workflow. Based on our collaboration with several research groups, streaming and collective operations are a commonality among various edge processing workflows. Streaming processing relies on data with no or few dependencies and performance is typically limited by I/O throughput. Collective operations require several large blocks of data to build the final result, whether it be the final reconstructed image or 3D tomogram, for example.

These operations can be thought of as a transition from the source, raw sensor data, which typically generates data at the highest rate but with lower data amount than what goes into a minimal data unit. The transition from the data generator (sensor) to the final data product (assembled data) is visualized in Fig. 2.

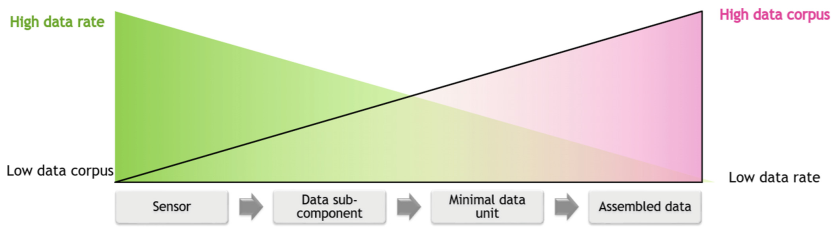

Fig. 2. Evolution of data rate and data corpus through a processing pipeline

A key factor in this data-rate and data-corpus transition is the hardware and software requirements and architecture at each step. When the peak data-rate is highest, ensuring that the data processing and streaming is not limited by I/O or inefficient processing is critical. When the data-corpus is high, I/O throughput is less critical, but processing times and potentially memory capacity become more critical and need to rely on larger computational resources.

3 Applications

With the transition from high-data rates to high-data corpus pipelines in mind, we will now introduce two application areas that fit this model, *ptychography* and *lattice light sheet microscopy*. For each of these applications, the specifics of the data flow components including data dimensions and rate for each step will show how the general concept applies and conforms to general computational workflows originating at scientific instruments.

Through representative examples from ptychography and lattice light sheet microscopy, we demonstrate the application of these technologies to enable real-time data processing and visualization at the edge.

3.1 X-Ray Ptychography

One application that we will focus on is ptychography, a growing technique using high-energy X-rays to image objects with high resolution in 2D and even 3D. The x-rays are created at a facility commonly called a “light source”, which derives the x-rays from a synchrotron. The technique eschews using a lens to focus the light, and instead a sensor collects a large number of scattered images, each associated with a different scan position (due to moving either the specimen or light source), as seen in Fig. 3. This enables sharper final images as well as energy regimes where lens design and construction is very challenging or impossible. This collection of images (which visually appear only as scattered light or inference patterns) undergoes several processing steps followed by a final iterative reconstruction stage which yields an image of the scanned object [7]. Depending on the location and angle of the light source, these final scans can be associated with a particular depth, such that a collection of scans can be processed into a full 3D tomographic image allowing for non-destructive visualization of integrated circuits [8] with a resolution down to 10 nm scale [9].

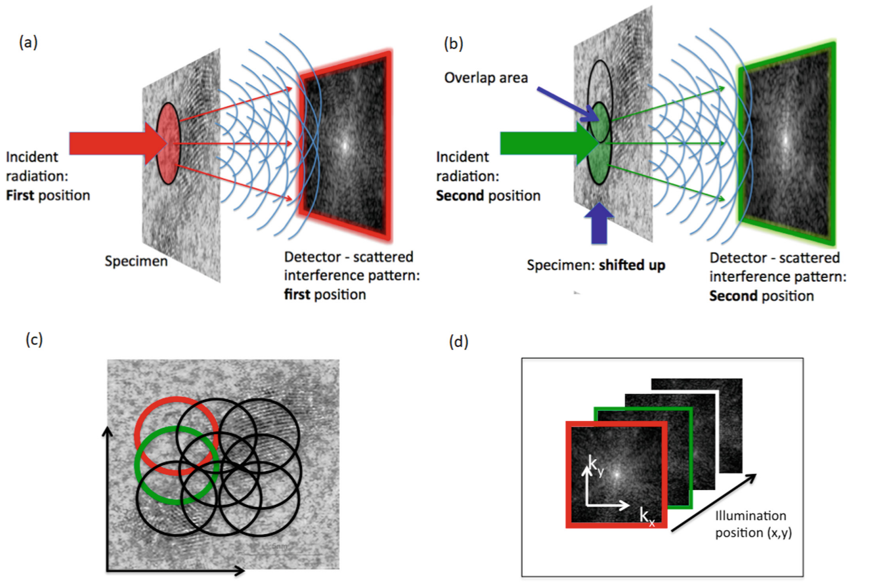
(a)
Incident radiation: First position
Specimen
Detector - scattered interference pattern: first position
(b)
Overlap area
Incident radiation: Second position
Specimen: shifted up
Detector - scattered interference pattern: Second position
(c)
(d)
k_y
k_x
Illumination position (x,y)

Fig. 3. Ptychography imaging overview (Image Credit: Wikipedia user 22sm22/CC BY-SA 4.0).

The data rate of ptychography is simply a product of scan rate (number of images/sec), the sensor resolution, and the image bit-depth. Current experiments produce raw data in the GB/s rate, but as sensor rates and resolutions increase, this is expected to increase to TB/s in the near term. A ptychography processing pipeline follows a common pattern:

1. Byte-level processing – arranging the raw sensor data into the image format
2. Scattered Image processing – combining multiple exposures, filtering, and image downsampling.
3. Image Reconstruction – Iteratively combine many scattered images into a single reconstructed image of the original object.

The image reconstruction is where the bulk of the algorithmic, scientific, and computational challenges are located and, accordingly, where the bulk of the development work has focused. The reconstruction algorithm itself is mostly limited by FFT throughput and performance, which is particularly well suited to GPU-accelerated FFT libraries like CUFFT. In our experience, reconstruction implementations are written using CuPy [13], PyCuda [10], and C++ with CUDA [11] and can leverage MPI to accelerate processing on multiple GPUs across many processing nodes [3,11,12]. Although most current reconstruction implementations currently require all scanned images to be present, the iterative nature of the algorithm does allow for streaming reconstruction [6], meaning that a progressive and real-time visualization of the result is possible.

Many of the codes referenced here have seen their initial processing bottleneck from step 3 to a combination of all three steps, along with any downstream processing (after step 3). This performance bottleneck is typically due to 3 factors:

1. Single-threaded Python & NumPy (CPU-only) byte and image processing.
2. Disk & File throughput limitations.
3. "Keyboard throughput" i.e., limits for users to coordinate the processing by copying files and starting scripts on a sequence of machines.

Factor 1 can be accelerated by converting the NumPy code to JAX, which we highlight in Sect. 4. Factors 2 & 3 are a more complex challenge. Files are commonly the conduit between stages, and users initiate and manage the pipeline stages by hand, transferring files from acquisition to compute nodes for preprocessing and again to multi-node systems for final analysis and reconstruction. We solve this problem in Sect. 5, using NVIDIA's SRF streaming pipeline framework.

3.2 Lattice Light Sheet Microscopy

Lightsheet microscopy is used for 3D high resolution imaging of biological samples with minimal phototoxicity and photobleaching. Images are obtained by illuminating portions of samples in the focal-plane with thin sheets of light. The fluorescence from the molecules excited within each optical section, and the field of view of the observing lens, are collected and stacked. Multiple color channels of the image correspond to different wavelengths of the light.

The data rate depends on the experiment at hand. For living and evolving biological samples, multiple image volumes are collected in short burst cycles. For dead specimens, large single volume images are acquired. Automating data acquisition will enable researchers to obtain reproducible results with minimal manual intervention, increasing experimental throughput and reliability. As a result, new techniques should be implemented to discover unique biological events. For example, cancer cells were observed splitting from 1 cell to 3 cells, instead of a canonical 1-to-2 split. This is seen in Fig. 4(a) in experimental output from the Advanced Bioimaging Center, University of California, Berkeley.

A typical data processing pipeline (Fig. 4(b)), among other processing steps, involves

1. Light sensitive camera captures a series of frames corresponding to the volume of the physical sample
2. Iterative deconvolution to filter out noise and undo the transfer function of the optical instrument
3. Deskew the 3D image volume to orient the image with respect to instrument coordinates
4. Visualizing the processed image volume in instrument coordinates

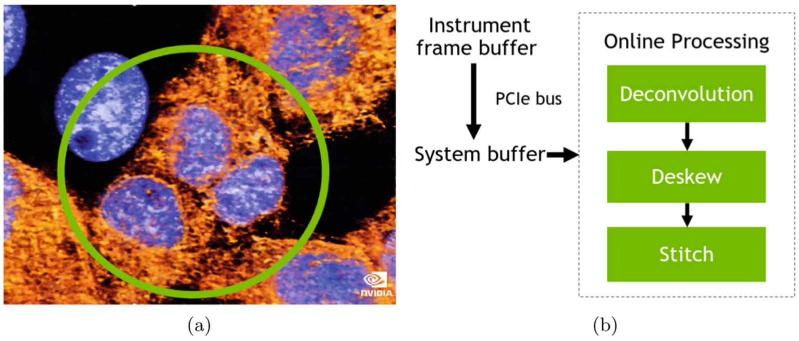

Fig. 4. (a) 3-way splitting of cancer cells (b) Typical processing steps in lightsheet microscopy

For this pipeline, visualization is desired not only as the final result, but also at several intermediate steps in the processing pipeline. A pipeline framework (see Sect. 5) can help break up the individual operations and expose the data as it moves through the pipeline to enable this visualization without a significant overhead or added implementation complexity.

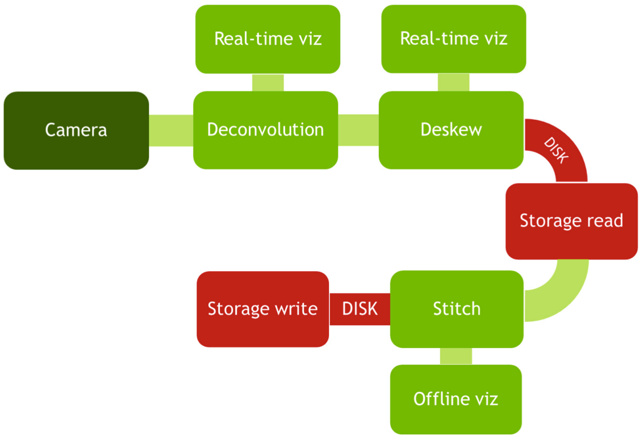

Fig. 5. SRF segment and nodes for real-time data processing. Visualization is provided for each operation of the processing pipeline.

4 High Performance GPU-Enabled Python

Working with the ptychographic imaging group at the Advanced Light Source (ALS), we were able to profile and help optimize their processing pipeline. As mentioned earlier, they put significant effort into optimizing their reconstruction software, leveraging CuPy to enable GPU-based computing, and MPI to accelerate the time-to-solution as well as enable larger working image sets. Without this acceleration, the reconstruction would be the stand-out bottleneck, but as can be seen in Fig. 6, for an example of 2500 scans of 1040×1152 resolution, the image processing (pre-processing) is the obvious next optimization goal (Fig. 5).

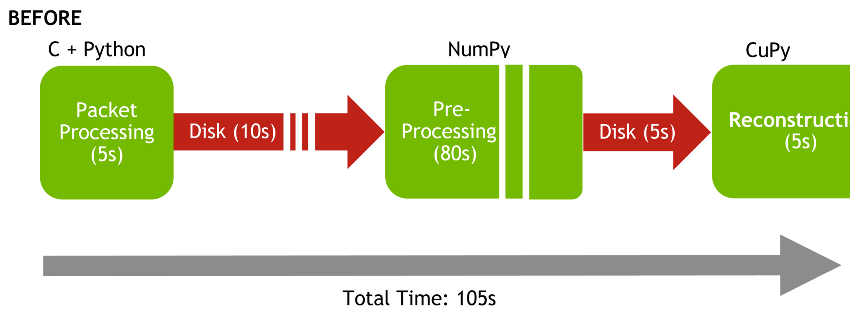

Fig. 6. Initial ptychographic processing pipeline with timing information

The pre-processing was written in NumPy with HDF5 (h5py) as the file storage library. The many NumPy operations in this processing step made it an ideal candidate for JAX, a python-based computing library from Google with CPU, GPU, and TPU support through their XLA compiler. Compared to another GPU-enabled framework like CuPy where all numerical expressions are strict, it has the ability to trace expressions at the function level at runtime. The intermediate representation created by each functional trace is passed to the XLA compiler for just-in-time compilation (jit) or other analysis like batch processing (vmap), creation of expression gradients (grad), or multi-GPU parallelization (pmap).

Superficially, porting NumPy to jax can be as simple as replacing
`import numpy as np`
with
`import jax.numpy as np`.

In practice, however, to enable all tracing features, jax enforces variable immutability. Thus any places where variables are updated using indexing, require a change in syntax from
`imgs_out[i,:,:] = process(imgs_in[i,:,:])`
to
`imgs_out = imgs_out.at[i].set(process(imgs_in[i,:,:]))`
Also any significant loops need modification to use JAX's looping mechanism so that JAX can capture the semantics of the loop without unwinding all expressions at runtime. Finally, h5py provides a convenient NumPy-like interface into the arrays stored in the HDF5 file, such that file access is on-demand and appears

to the user as simple numpy arrays. These file-access objects must be explicitly moved into JAX arrays on the GPU.

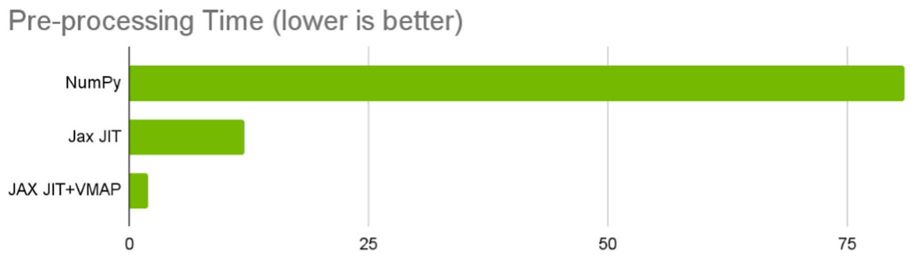

Fig. 7. Timing information for pre-processing using JAX `jit()` and `vmap()`.

Figure 7 highlights how leveraging JAX's features accelerates the preprocessing step. After applying the aforementioned changes to the processing code, including only the `jit()` operation, we already saw a 7x performance improvement from 80 s to 12 s. With some additional work to capture all processing into a single function called `def process(image)` which has only a single image as input (tensor shape = (1040, 1152)), we can leverage JAX `vmap()` operator to create a function that processes all the images (tensor shape = (2500, 1040, 1152)), which improved the runtime from 12 s to 2 s, another 6x gain for a total gain of 40x.

We additionally highlight that despite being ported to JAX, the code is still (mostly) NumPy processing code, which can be read and modified by the application scientists who first wrote and continue to maintain the code. This is a critical requirement; legibility and modifiability is critical for these smaller scale HPC applications where the user-base is small. This motivates the next section of the paper — how to chain these frequently python-based processing nodes together without losing performance (GPU data stays on the GPU) while maintaining legibility for the users.

5 Streaming Processing Pipelines

The component-wise speedups in the JAX work shown in the previous section are a critical component of the performance story of a processing pipeline, but will expose the inefficiencies in an existing file-based pipeline. Copies between CPU host memory and GPU device memory create performance bottlenecks, and using stored files as the conduit between pipeline stages only exacerbates the bottlenecks. If users manage the pipeline stages manually, often the latency they introduce will be another bottleneck, and a source of errors.

To maintain the performance gained through the use of GPU-computing we require a framework that allows us to create computational pipelines that have the following capabilities:

1. GPU-aware: Data on the GPU stays on the GPU when possible
2. Network aware: Transparent (to the user) high-performance transferring of data between physical nodes is critical because most pipelines will reach from the edge to the computing center.
3. Easy to build and maintain: Building pipelines in Python (with GPU-enabled Python frameworks) should feel natural. Additionally debugging and profiling should be possible with standard tools.
4. High Performance: Overhead should be minimal and pathways to "upgrade" pipeline stages from Python to C++ should feel natural.

Introducing SRF. Streaming Reactive Framework (SRF) is a component of NVIDIA's Morpheus (a network analysis software-development kit (SDK)) that allows users to build high-performance streaming data pipelines. It supports building complex pipelines that involve branching, joining, flow control, feedback, and back pressure. The sequence of data processing operations is captured in a computational graph. Visualized in Fig. 8, the basic building blocks of this computational graph are called nodes and segments. SRF-nodes define basic computational units, typically python functions, that perform computationally expensive operations on an input to produce an output. The connectivity of the computational graph is broken into "segments", the SRF-nodes of which are guaranteed to run on the same compute resource, meaning that node-to-node transfers remain in GPU or system memory. For segments executing on different computational resources, data transfer occurs through the network. SRF orchestrates the execution of this data pipeline by setting up an event-loop, asynchronously offloading the computation and efficiently executing the processing pipeline on the available compute-resources.

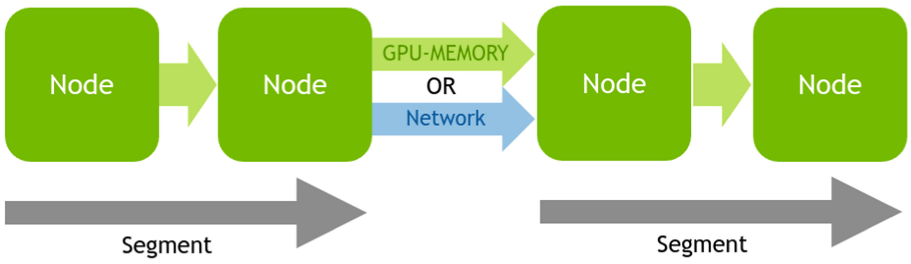

Fig. 8. SRF pipeline with several nodes connected by two segments.

SRF has a C++ runtime and the nodes typically run within a single process, and hence common debugging and profiling tooling will "just work". If several segments connect across multiple processes on different nodes, the standard tooling needs to be adapted accordingly. Nodes within SRF can be written either in Python or C++. C++ nodes are type-checked for compatibility. Python nodes simply transfer python objects from one node to the next and hence the ownership has to be managed by the user i.e., care must be taken to avoid modifying data downstream.

In Python, defining a segment is as simple as defining a source (see code below), processing node, and a sink (or just source & sink). In the code below, you can see an example of defining such a pipeline, where `deconvolve()` implements an image processing algorithm.

```python
def segment_init(seg: srf.Segment):
    source = seg.make_source("source", data_source())
    sink = seg.make_sink("sink",
                         sink_on_next,
                         sink_on_error,
                         sink_on_complete)

    deconvolution_node= seg.make_node("deconvolution",
                                      lambda x: deconvolve(x, PSF))
    seg.make_edge(source, deconvolution_node)
    seg.make_edge(deconvolution_node, sink)
```

A key usability feature of SRF is that nodes pass Python objects between each other, such that a deconvolution node simply wraps your existing deconvolution routine without changes, as can be seen here.

```python
def deconvolve(img, psf, iterations=20):
    '''
    This function runs the Richarson-Lucy deconvolutions.
    :img: Is the input image and it could be a numpy array or a cupy array.
    :psf: Is the Point Spread Function and it can be a numpy array or a cupy array.
    :iteration: Is the number of Richarson-Lucy iterations.
    '''

    # Pad PSF with zeros to match image shape
    pad_l, pad_r = np.divmod(np.array(img.shape) - np.array(psf.shape), 2)
    pad_r += pad_l
    psf = np.pad(psf, tuple(zip(pad_l, pad_r)), 'constant', constant_values=0)

    # Recenter PSF at the origin.
    for i in range(psf.ndim):
        psf = np.roll(psf, psf.shape[i] // 2, axis=i)

    # Convolution requires FFT of the PSF
    psf = np.fft.rfftn(psf)

    # Perform deconvolution in-place on a copy of the image (avoids changing the original)
    img_decon = np.copy(img)
    for _ in range(iterations):
        ratio = img / np.fft.irfftn(np.fft.rfftn(img_decon) * psf)
        img_decon *= np.fft.irfftn((np.fft.rfftn(ratio).conj() * psf).conj())
    return img_decon
```

As we saw in the JAX section, batching is a critical performance optimization in many image-processing algorithms. We can implement a batched node in a fashion similar to the code below:

```python
def batch_pipeline(pipe_state, metadata, frame_in, frame_out):
...
def on_next(frame):
    i_batch = pipe_state["current_num_in_batch"]
    pipe_state["exp_frame_batch"][i_batch, :, :] = frame_i
    pipe_state["current_num_in_batch"] += 1
    if pipe_state["current_num_in_batch"] == metadata["batchsize"]:
        pipe_state["current_num_in_batch"] = 0
        return pipe_state["exp_frame_batch"]

def on_complete():
    if pipe_state["current_num_in_batch"] > 0:
        batch_i = pipe_state["current_num_in_batch"]
        return pipe_state["exp_frame_batch"][:batch_i, :, :]
        frame_in.pipe(ops.map(on_next),
                      ops.filter(lambda x: not isinstance(x, type(None))),
                      ops.on_completed(on_complete)).subscribe(frame_out)
```

By maintaining a mutable state variable `pipe_state`, we gather frames into a batched image container, which is passed downstream to the next node once full (`on_next`). If the stream ends without completely filling the batch, `on_complete` sends the existing frames which ensures that we neither hang waiting for more frames nor drop frames. This enables a pipeline seen in Fig. 9. Considering the

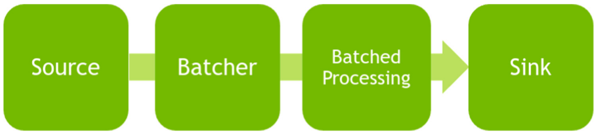

Fig. 9. Batched deconvolution pipeline

pre-processing example given in the JAX section, with this pipeline we are able to eliminate the disk stages taking the runtime from 105 s to 12 s, a speedup 9x which is highlighted in Fig. 10. By using SRF to eliminate files connecting the

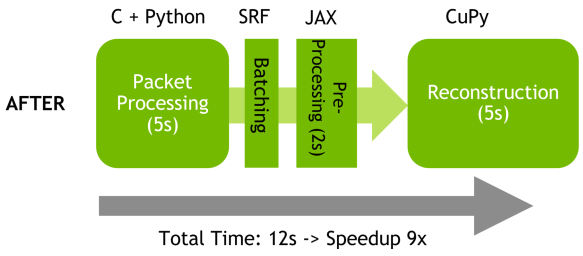

Fig. 10. Batched Ptychographic processing pipeline using JAX & SRF.

three pipeline stages we can yield a total speedup of 9x. Additionally, we can run the packet processing and image pre-processing at a node close to the sensor, and the reconstruction in a more-powerful system with pre-processed images transferred over the network.

5.1 Supercomputing as a Service (SCaaS)

With SRF, we can process and stream data from high-throughput edge hardware to larger-compute resources located at a supercomputing center to handle the high data corpus components of the processing workflow (e.g., ptychographic reconstruction) where time-to-solution and memory requirements are more demanding. With most of the processing software challenges solved, the problem becomes the ability to guarantee computing resources during data collection and stream the data from the experiment to the cluster. Recent work towards streaming Ptychography process in Switzerland at the Paul Scherrer Institute, (PSI) and the Swiss Supercomputing Center (CSCS) have managed to build streaming workflows [11] and have proven that these centers can stream results reliably and securely.

Alternatively, Globus (globus.org) is a framework and system for managing large datasets, and is a common way for scientists to transfer, manipulate, and process files across their own local systems and supercomputing resources like those found at NERSC and others. Extending Globus is *Globus Automate*, which provides the ability to write "flows" which allow the user to specify computational events triggered by files. These computational stages can be run locally or on supercomputing resources, yielding a *service* oriented structure to the computation and the center itself [4]. Assuming Ptychography or other Sensor-driven workflows are pipelined with SRF, we also require "SCaaS" to enable data to be streamed to the computing center live while avoiding expensive file-oriented designs, which limit the streaming performance to disk throughput.

The framework and the GPU optimizations highlighted in this paper have given solutions to the software architecture and performance requirements for streaming sensor processing, however we hope that supercomputing centers can use it as a benchmark for modifying and designing current and next generation systems for the expanding needs of their users.

6 Conclusion

Instruments and their research tasks have unique workflows, but have overall commonalities where best practices can be applied. Examples as diverse as particle accelerator beam lines to constantly changing novel light microscopes, we apply accelerated processing and visualization where it is most beneficial today. Each scientific domain will adopt such techniques independently.

Science and manufacturing disciplines are deploying more dynamic approaches to increase the efficiency of instrument usage: maximize data quality and minimize blind data acquisition. Developing integrated intelligent workflows as close to the sensor as possible ultimately improves data acquisition and optimizes time per experiment [14]. The demand is increasing for instruments like cryo-em, which make extensive use of automation already, but high purchase and operation costs are barriers to access [2]. National centers like NIH-funded cryo-em centers [1] are the worldwide trend. The focus is on maximizing instrument throughput and training new technologists. Globally such efforts democratize access to expensive instruments and technological best practices.

A key component to ensuring these trends continue is rapid access to data at all steps of the acquisition process. Breaking down the data acquisition process into components from raw data streams to final assembled data gives us an opportunity to apply optimal processing at each step using the right level of computational infrastructure. With access to the data, streaming visualization can be more easily implemented, giving quicker feedback for experimental adjustments and even higher-level experimental control. The GPU-accelerated Python and SRF tooling described in this paper show the advantages such workflows could provide to help the application scientists achieve the performance necessary to unlock the potential of their newest and future devices while keeping their software simple enough for them to maintain.

Acknowledgements. We thank our colleagues Jack Wells, Chris Porter, and Ryan Olson for their useful feedback. We additionally thank David Shapiro and Pablo Enfedaque at ALS for their collaboration; and Gokul Upadhyaula, Matthew Mueller, Thayer Alshaabi, and Xiongtao Ruan at the Advanced Bioimaging Center, University of California, Berkeley for their ongoing collaboration.

References

1. NIH funds three national cryo-EM service centers and training for new microscopists | National Institutes of Health (NIH). https://www.nih.gov/news-events/news-releases/nih-funds-three-national-cryo-em-service-centers-training-new-microscopists
2. The must-have multimillion-dollar microscopy machine | News | Nature Index. https://www.natureindex.com/news-blog/must-have-multimillion-dollar-microscopy-machine-cryo-em
3. Batey, D., Rau, C., Cipiccia, S.: High-speed X-ray ptychographic tomography. Sci. Rep. **12**(1), 1–6 (2022). https://doi.org/10.1038/s41598-022-11292-8
4. Blaiszik, B., Chard, K., Chard, R., Foster, I., Ward, L.: Data automation at light sources. In: AIP Conference Proceedings, vol. 2054, no. 1, p. 020003 (2019). https://doi.org/10.1063/1.5084563. https://aip.scitation.org/doi/abs/10.1063/1.5084563
5. Elbakri, I.A., Fessler, J.A.: Statistical image reconstruction for polyenergetic X-ray computed tomography. IEEE Trans. Med. Imaging **21**(2), 89–99 (2002). https://doi.org/10.1109/42.993128
6. Enders, B., et al.: Dataflow at the COSMIC beamline - stream processing and supercomputing. Microsc. Microanal. **24**(S2), 56–57 (2018). https://doi.org/10.1017/S1431927618012710. https://www.cambridge.org/core/journals/microscopy-and-microanalysis/article/dataflow-at-the-cosmic-beamline-stream-processing-and-supercomputing/2F4AD3721A36EE02C0336A8191356065

7. Guizar-Sicairos, M., et al.: High-throughput ptychography using Eiger: scanning X-ray nano-imaging of extended regions. Opt. Express **22**(12), 14859–14870 (2014). https://doi.org/10.1364/OE.22.014859
8. Holler, M., et al.: High-resolution non-destructive three-dimensional imaging of integrated circuits. Nature **543**(7645), 402–406 (2017). https://doi.org/10.1038/nature21698. https://www.nature.com/articles/nature21698
9. Holler, M., et al.: Three-dimensional imaging of integrated circuits with macro- to nanoscale zoom. Nat. Electron. **2**(10), 464–470 (2019). https://doi.org/10.1038/s41928-019-0309-z. https://www.nature.com/articles/s41928-019-0309-z
10. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput. **38**(3), 157–174 (2012). https://doi.org/10.1016/J.PARCO.2011.09.001
11. Leong, S.H., Stadler, H.C., Chang, M.C., Dorsch, J.P., Aliaga, T., Ashton, A.W.: SELVEDAS: a data and compute as a service workflow demonstrator targeting supercomputing ecosystems. In: Proceedings of SuperCompCloud 2020: 3rd Workshop on Interoperability of Supercomputing and Cloud Technologies, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 7–13 (2020). https://doi.org/10.1109/SUPERCOMPCLOUD51944.2020.00007
12. Marchesini, S., et al.: SHARP: a distributed GPU-based ptychographic solver. J. Appl. Crystallogr. **49**(4), 1245–1252 (2016). https://doi.org/10.1107/S1600576716008074. http://scripts.iucr.org/cgi-bin/paper?jo5020. URN: ISSN 1600-5767
13. Okuta, R., Unno, Y., Nishino, D., Hido, S., Loomis, C.: CuPy: a NumPy-compatible library for NVIDIA GPU calculations. Technical report (2017). https://github.com/cupy/cupy
14. Zhang, Z., et al.: Toward fully automated UED operation using two-stage machine learning model. Sci. Rep. **12**(1), 1–12 (2022). https://doi.org/10.1038/s41598-022-08260-7. https://www.nature.com/articles/s41598-022-08260-7
15. Zheng, S.Q., Palovcak, E., Armache, J.P., Verba, K.A., Cheng, Y., Agard, D.A.: MotionCor2: anisotropic correction of beam-induced motion for improved cryo-electron microscopy. Nat. Methods **14**(4), 331–332 (2017). https://doi.org/10.1038/nmeth.4193. https://www.nature.com/articles/nmeth.4193