JOURNAL OF SYNCHROTRON RADIATION

ISSN 1600-5775

Real-time diffraction computed tomography data reduction

J. Kieffer,* S. Petitdemange and T. Vincent

ESRF – The European Synchrotron, CS40220, 38043 Grenoble Cedex 9, France.
*Correspondence e-mail: jerome.kieffer@esrf.fr

Received 28 November 2017
Accepted 10 January 2018

Edited by A. F. Craievich, University of São Paulo, Brazil

Keywords: powder diffraction; diffraction imaging; Pilatus detector; image compression; azimuthal integration; pyFAI; FabIO; silx.

Diffraction imaging is an X-ray imaging method which uses the crystallinity information (cell parameter, orientation) as a signal to create an image pixel by pixel: a pencil beam is raster-scanned onto a sample and the (powder) diffraction signal is recorded by a large area detector. With the flux provided by third-generation synchrotrons and the speed of hybrid pixel detectors, the acquisition speed of these experiments is now limited by the transfer rate to the local storage as the data reduction can hardly be performed in real time. This contribution presents the benchmarking of a typical data analysis pipeline for a diffraction imaging experiment like the ones performed at ESRF ID15a and proposes some disruptive techniques to decode CIF binary format images using the computational power of graphics cards to be able to perform data reduction in real time.

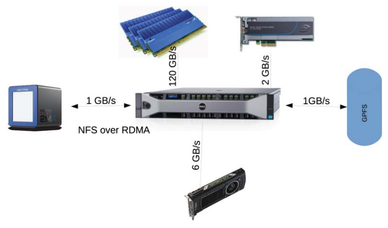
1. Introduction

Since all major third-generation synchrotrons are undergoing upgrades to provide brighter sources (Biasci *et al.*, 2014; Tanaka, 2014) the same flux of photon will be available soon in much smaller beams. Two types of experiments will benefit the most from this improved X-ray source: coherence diffraction experiments and raster-scanning experiments.

X-ray diffraction computed tomography (hereafter XRD-CT, Fig. 1) (Jacques *et al.*, 2011) is one of the raster-scanning experiments where a pencil beam is scanned onto a sample. The volumetric information is obtained by rotating the sample in the X-ray beam to generate the sinogram. The diffraction signal, scattered over a large solid angle, is recorded by an area detector and saved as a stack of images. Those images are azimuthal averaged into powder diffraction patterns with the intensity given as a function of either the diffraction angle (2θ) or the scattering vector $q = 4\pi\sin(2\theta/2)/\lambda$. The sinogram is built, pixel by pixel, by storing this pattern as a function of the sample position: translations and rotation. The tomogram is finally reconstructed from the back-projection of the sinogram.

It turns out that the current detectors are already fast enough to fill the temporary storage, and data analysis workflows cannot cope with the pace imposed by modern detectors (Mokso *et al.*, 2017): data analysis *is* the limiting factor for the whole experiment. This work focuses on the performance optimization of the data reduction pipeline for cases where the reduction is simply an azimuthal regrouping of input images. More complex analyses are possible (and often desirable) but the target of this work is online data analysis so the analysis will be restricted to simple ones.

We will focus in the second section on the setup of the materials beamline (ID15a) of the ESRF (Vamvakeros *et al.*,



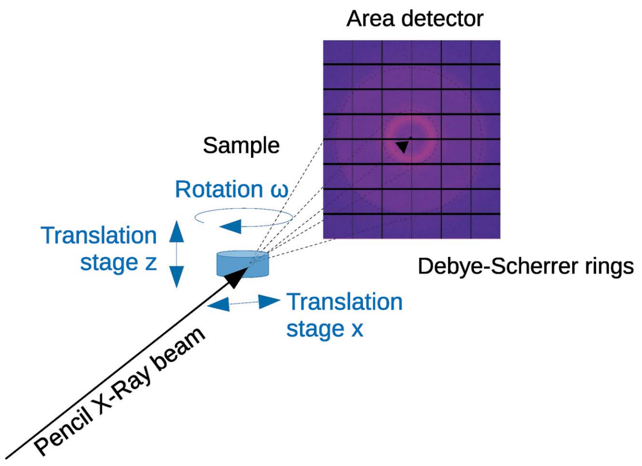


Figure 1
Scheme of an X-ray diffraction tomography setup where the sample can be translated and rotated in front of a pencil beam. The Debye–Scherrer rings formed by the diffraction of crystallites are collected by an area detector, integrated into a powder diffraction pattern and assembled to create a sinogram, which is computationally back-projected to build the tomogram.

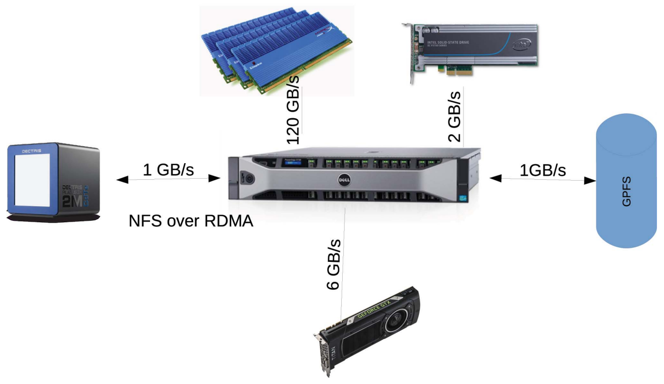


Figure 2
Data analysis computer with the main interconnections and their associated bandwidth.

2016) and perform a complete benchmarking of the data analysis pipeline used. This will highlight various bottlenecks in the data analysis chain. To address the image decompression bottleneck, different parallelization schemes have been developed and are presented in §3.

2. X-ray diffraction imaging: data analysis pipeline

2.1. Beamline hardware

2.1.1. Pilatus3 2M CdTe detector. The ID15a beamline at the ESRF uses mainly a Pilatus3 2M detector with a 1000 µm CdTe sensor, manufactured by Dectris (Kraft *et al.*, 2009). The detector is made up of 8 × 3 Pilatus modules (100 kilopixels each). Unlike silicon-based sensors, there are two Cd–Te wafers bump-bound to every single Pilatus module, with a gap of 3 pixels between the wafers. The gaps between the Pilatus modules are the same as in the silicon-based detectors, *i.e.* 7 pixels vertically and 17 pixels horizontally.

This detector is sold with a detector-PC which is in charge of compressing and saving the images on the network. This detector-PC comes with a 10 Gbit s^{-1} network card and is directly connected to the data analysis server.

The detector is advertised as operating at 250 frames per second (fps). Each frame has 2.4 megapixels, stored as 32-bit integers (the dynamic range is only 20 bit). At full speed, the raw (uncompressed) stream thus represents 19.2 Gbit s^{-1} to transfer. Compression is hence mandatory to transfer the acquired data through the 10 Gbit s^{-1} network interface. Dectris uses the CIF binary format (CBF) (Bernstein & Hammersley, 2006) for Pilatus detectors with byte-offset compression which provides a compression factor close to 4×. An alternative compression scheme used in the novel Eiger detector is the LZ4 but the compression factor is much less, around 2× (and variable, depending on the signal of the detector) which makes this option not applicable for operating the Pilatus detector continuously over an extended period of time.

2.1.2. The data analysis computer. The data analysis computer (Fig. 2) acts as an NFS (network file server) server over RDMA (remote direct memory access) for optimal performance with the detector and is directly connected to the GPFS (general parallel file-system) storage cluster (Schmuck & Haskin, 2002). This data analysis computer has two Intel Xeon E5-2643 v3 processors, each with six cores and 20 MB of cache, and 128 GB of memory. There are additionally two 10 Gbit network cards, a fast Intel P3700 solid-state drive (SSD) and an Nvidia Titan-X graphics card, all connected on the PCI-express bus.

2.2. Processing for a diffraction imaging experiment

The pre-processing for diffraction imaging experiments is typically the azimuthal integration of the whole image with some mask. The code snippet in Appendix *A* is a typical example: each image is read, integrated in a one-dimensional profile and saved in a HDF5 file [here not following the Nexus (NIAC, 2003) convention for the sake of concision in this example].

This code snippet is written in the Python programming language (van Rossum, 1989) and uses some extra libraries:

(i) H5Py: for accessing HDF5 files in Python (Collette, 2013).

(ii) FabIO: for reading most of the X-ray image formats (Knudsen *et al.*, 2013).

(iii) PyFAI: for performing the azimuthal integration from two-dimensional images into one-dimensional profiles (Kieffer & Karkoulis, 2013).

The function 'process' described in this code snippet has been used for profiling the application (*i.e.* measuring how much time is spent in each part of the code). The input data set is composed of 1202 images in CBF format coming from an actual experiment on ID15a stored on the SSD. One should distinguish two cases: when data are only available on the solid-state disk and when they are available in the cache of the

Table 1
Average performance (in fps) for the azimuthal integration of 1202 Pilatus3 2M CdTe images saved in CBF format read from SSD (or from memory) on a dual Xeon E5-2642v3 with an Nvidia Titan-X graphics card.

Median over five runs and median deviation to the median.

Strategy	Readers	Integrator	Data source	Median speed (fps)	Deviation (fps)
Sequential	CPU	CPU	SSD	28.2	0.3
Sequential	CPU	GPU	SSD	53.1	1.4
Sequential	CPU	CPU	MEM	31.6	0.9
Sequential	CPU	GPU	MEM	74.6	1.3
Pool	1	GPU	MEM	167	3
Pool	2	GPU	MEM	192	7
Pool	3	GPU	MEM	155	5
Pool	4	GPU	MEM	190	4
Pool	6	GPU	MEM	168	18
Pool	12	GPU	MEM	121	4
Pool	24	GPU	MEM	88	2
OpenCL	GPU	GPU	MEM	253	2

operating system. The total amount of raw data is 3 GB, so only the first read can be considered as a 'cold-start'; subsequent reads actually benefit from the cache of the operating system and should be considered as a 'hot-start'. To be able to profile in 'cold-start', the disk has been un-mounted and re-mounted to flush all caches. Each measurement has been performed five times and the results are reported in Table 1. Only the median frame rate with the median absolute deviation to this median have been reported in fps (or Hertz).

The first and the third (respectively, second and fourth) lines report the performance in cold- and hot-start. This allows us to evaluate the actual read time per frame from the SSD drive which is 4 ms (respectively, 5 ms).

As a consequence, it is impossible to process images at 250 fps from this SSD as it is not possible to read the (compressed) data at the required pace. There is an emerging technology (3D XPoint technology by Intel) for replacing NAND cells in SSDs which looks promising and should offer lower latency for acting as a cache for the raw data. As of today, those drives are not yet available in capacities large enough for replacing the memory for the kind of temporary storage needed for beamline application.

As online data analysis has to rely on data 'living in memory' and not read from any drive, the 128 GB of memory available on the computer represents a cache of about 3 min of experiment time and thus the data analysis pipeline has to keep up the pace of the experiment. Profiling 'precisely' the data analysis program is hence of crucial importance.

2.3. Profiling of the data analysis pipeline

The snippet of code from Appendix *A* presented previously has been run in a profiler to measure how much time is spent in every part of the code for a reference data set of 1202 compressed images. The code executes on this reference data set in 43 s and the three most time-consuming parts are: the azimuthal integration (29 s), the byte-offset decompression (4.9 s) and the checksum calculation (4.7 s). The checksum verification is optional in CBF and may simply be ignored.

The azimuthal integration was originally performed on the processor and could be offloaded to the graphics card, which lowers azimuthal integration time to 13 s. As described by Kieffer & Ashiotis (2014), most of the time for azimuthal integration on a graphics card is spent on the transfer of the raw image to the device. To speed up the azimuthal integration, the best option would be to transfer less data to the graphics card, *i.e.* the compressed data, and decompress them on the GPU. Until now, this approach has still been challenging for unmodified compressed formats (Sitaridi *et al.*, 2016).

The bottleneck of byte-offset decompression remains; this algorithm will be described in detail in the next section and analysed.

3. Optimizing the decompression of CBF images

3.1. Decompression on the processor

The core idea of the byte-offset compression is to encode only the difference value between two adjacent pixels and hope this value is small enough to fit in an 8-bit (signed) integer, *i.e.* in the range −127 to +127. Larger values are coded with a special value (−128) which indicates an exception and the subsequent 2 bytes are decoded as a 16-bit integer in little-endian order. If the value does not fit in a 16-bit integer, a 32-bit exception is signalled (with value −32768) and the actual value is coded over the next 4 bytes as a 32-bit little-endian integer. Hence, each value can be coded with a variable size of 1, 3 (= 1 + 2) or 7 (= 1 + 2 + 4) bytes, which makes it very difficult to decompress in a parallel fashion.

Since 2004, processors have been running at a maximum speed of about 4 GHz. This means that a serial algorithm like the byte-offset decompression described previously runs at the same speed on a high-end computer as it would on a 13 year-old computer. Parallelization is the only way to get the processing done faster and a couple of strategies have been explored and will be presented.

3.2. Pool of workers strategy

A classical strategy in parallel computing is to attribute one type of computation to a given compute engine. With azimuthal averaging already being executed on the graphics card, it is natural to devote the image decompression to the central processor (CPU).

Unlike lower-end (disk) drives, where the data access is serialized, the SSD used in this experiment is interfaced in PCI-express using the NVME protocol (Xu *et al.*, 2015) which allows thousands of parallel accesses for reading and writing. We validated that the performance is actually better when the read step is performed with multiple threads rather than sequentially on this hardware.

A pool of workers (Fig. 3) is set up using multiple threads for reading and decoding the data. The number of workers in this pool is the parameter that needs to be optimized

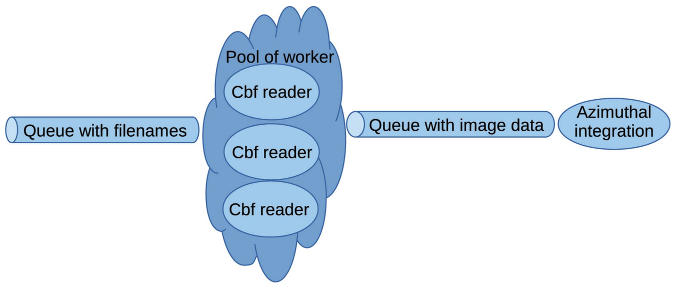


Figure 3
Workflow for the data reduction of an XRD-CT experiment using the pool of workers pattern.

depending on the computer, especially as a function of the number of processors, of cores and the amount of cache available. The list of files to be processed is distributed to the pool of readers *via* a parallel queue. The code snippet used for profiling is reproduced in Appendix *A*. Each worker, which is implemented as a thread, loops over input filenames it gets from the input queue. This queue guarantees that each file is processed once, regardless of which worker does the job. After reading the file and decompressing the data, each worker puts the image into the output queue. Later on, azimuthal integration and data saving, using HDF5, are again performed sequentially. As the order of the filenames in the input queue can be different from the order of the images in the output queue, due to parallel processing, it is important to propagate the index associated with each filename or frame.

Table 1 provides the number of frames processed per second when processing the sample set of frames and varying the size of the pool of readers: 1, 2, 3, 4, 6, 12 and 24 workers, all data being already in memory (hot-start). This number has to be compared with the frame rate of the detector of interest: 250 fps. The performance of the linear pipeline is given as a comparison; in this case the data can be read from the disk (Intel P3700 SSD, cold-start) or are already available in the memory, cached by the operating system (Linux).

It is noticeable that the optimal performance is reached with a number of workers in the pool much lower than the number of cores of the computer: two or four readers is optimal while the computer has 2 × 6 cores. There are multiple reasons for this:

(i) The main thread is also working: it controls the azimuthal integrator and the saving of the reduced data in HDF5 format.

(ii) The amount of cache of each processor, which is 20 MB, has to be compared with the 2.5 + 10 MB for each frame (encoded + decoded). More readers means less CPU cache for each of them which is detrimental to the performance.

(iii) Python threads are serialized *via* the 'global interpreter lock', called GIL. While most of the processing performed is done in the 'no-GIL' section, more threads makes it more likely that they will be fighting each other for acquiring the GIL.

It is frustrating to have a powerful parallel processor on the graphics card and see the total performance limited by the file decompression which is purely serial.

3.3. Parallel decompression of CBF images

This section gives an overview of an implementation of the byte-offset decompression on massively parallel processors like graphics processors (GPU) using the OpenCL (Stone *et al.*, 2010) programming language. In those devices, threads are grouped in 'work groups' of a given size. All threads from a work group perform the same sequence of operations on different data (in a single instruction, multiple data way). This is called a 'kernel' and looks like a function written in C language.

Parallel decompression of byte-offset may be divided into the following steps (Fig. 4), each of them being divided into one or two kernels called subsequently:

(i) Search for exceptions in the input stream (marked with the value −128) and register them. Other 'normal' pixel-to-pixel difference values are simply decoded and stored.

(ii) Process all registered sections of contiguous exceptions in parallel to decode them and store their value. Here the work-group size is one, so a single thread is processing a complete section of adjacent exceptions and multiple threads

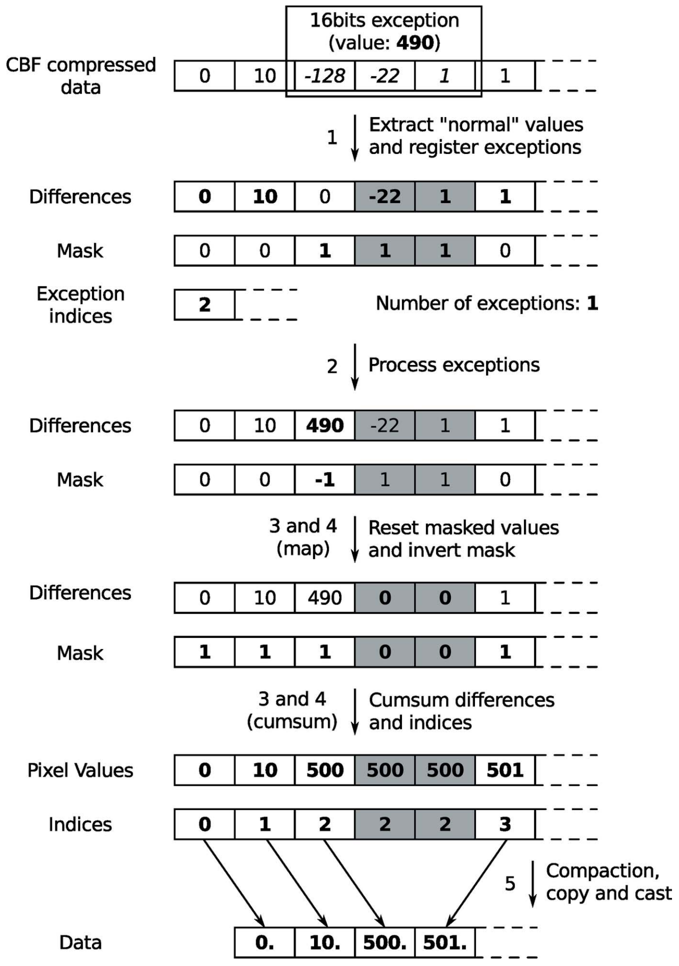


Figure 4
CBF decompression in parallel: step-by-step description of input and output buffers for each step (kernel). Bold values indicate values that have been changed in the previous step. The grey region corresponds to values longer than 1 byte which need to be sieved out in in the final step.

are processing multiple sections in parallel. If a thread starts in the middle of a section of contiguous exceptions, it does nothing as this section has to be processed by the thread which starts at the beginning of this section of contiguous exceptions.

(iii) Compute the cumulative sum of previously stored pixel-to-pixel difference values. This is performed using a 'prefix-sum algorithm' (Blelloch, 1989).

(iv) Compute the position of each value (in the output array) using the prefix-sum algorithm: a valid pixel position is set to one in the input and the other remains at zero. This algorithm provides the output pixel position for any input position. Technically the two prefix-sums are performed simultaneously in our implementation for better performance.

(v) Copy the reconstructed values at the proper place in the output array, optionally with a conversion to floating-point value to ease subsequent processing.

The implementation of this algorithm is available as part of the silx (Sole *et al.*, 2015–2017) library and will be part of the version 0.7. While GPUs targeted by this implementation have thousands of cores, this algorithm remains valid regardless of the number of cores. Our implementation, based on pyOpenCL (Klöckner *et al.*, 2012), has been validated on different architectures like Nvidia GPUs, integrated graphics processors (Intel Iris) found in laptops and multi-core processors (Intel, AMD and Apple OpenCL drivers).

The strength of this approach resides in the limited amount of data to be sent to the GPU memory, which allows decompression and integration to be performed on the device without additional transfer over the PCI-express bus (which is the bottleneck for pyFAI).

The performance of this parallel decompression of CBF images on a high-end GPU has been compared with the serial implementation. For large images that do not fit in the cache memory of the processor (typically for the Pilatus 6M images), the speed-up of this GPU version is important (10×). For smaller images, where data fit in the processor cache, the serial algorithm performs very well on the CPU, and hence the speed-up of the parallel version is rather limited (+50%). Nevertheless, the combination of the parallel CBF decompression with azimuthal integration on the graphics card allows us to exceed the 250 fps imposed by the detector as reported on the last line of Table 1. Moreover, this parallel implementation naturally benefits from advances in graphics card processors (*i.e.* more cores and faster memory): the same benchmark has been performed on a desktop computer with only one processor (Xeon 1650v3) instead of two, with less cache (15 MB instead of 20 MB), half the memory (64 GB instead of 128 GB) and a more recent, cheaper graphics card (Nvidia GTX 1080Ti instead of Titan-X). This desktop computer out-performed the server with more than 300 fps for this benchmark. This parallel algorithm is not only faster than the serial version, it is also much more stable in performance as the variability (expressed as the median of absolute difference to the median value) is only 2 fps on the GPU and twice more for the pool of workers pattern. The stability of the performance on the GPU can be explained by the dedication of the graphics card to this calculation.

4. Outlook

To be able to interconnect the decompressed data obtained in the silx library with the azimuthal integrator provided by pyFAI, without transferring the data back and forth from the device to the processor, the silx team implemented a way to exchange memory objects located on the graphics card between libraries. These results show the validity of the concept which paves the way to interconnecting different algorithms including image analysis and tomography algorithms which are available as part of the silx library. In pyFAI, a couple of advanced statistical analysis tools, recently ported to OpenCL like median filtering in two-dimensional integrated data and sigma-clipped average, could also be good candidates for this kind of direct interconnection.

The other strength of this approach is that it hides completely the complexity of byte-offset decompression with a simple 'decompression object'. In the GPU-based code snippet (Appendix *A*) this 'decompression object' is called 'bo'. The GPU-base code snippet is equivalent to the sequential one (Appendix *A*); it is only a few lines longer and uses the same strategy.

5. Conclusion

Processors used for data analysis hit the power wall more then a decade ago. Since then, no noticeable increase in performance has been seen on sequential algorithms, causing a bottleneck in the processing pipeline for many beamlines, especially those doing diffraction imaging. To be able to cope with the stream of data coming from a modern detector, today's fastest SSD drives are not (yet) fast enough to act as an effective cache and the data should best be kept in memory. Two types of parallelization have been evaluated to speed up the processing. The 'pool of workers' strategy has been evaluated for reading and decoding different images in parallel on different cores. It provides additional performance compared with the serial implementation but the speed is not proportional to the number of cores of the computer, probably due to cache congestion.

The first parallel implementation of the byte-offset decompression algorithm is also presented. Leveraging the performance of recent graphics cards, this code allows the reduction of data for a diffraction imaging experiment performed at full speed with a Pilatus3 2M detector (250 Hz), in real time. Moreover this implementation is even faster on newer hardware.

APPENDIX *A*

Figs. 5 to 7 give code snippets used for profiling the performances of three approaches presented in this work. Fig. 5: sequential decompression and azimuthal integration. Fig. 6: pool of reader with a sequential integration. Fig. 7: GPU-based decompression and azimuthal integration.

```python
import h5py, fabio, pyFAI

def process(list_of_files, result_file, geometry_file, number_of_bins):
    ai = pyFAI.load(geometry_file)
    with h5py.File(result_file) as result:
        dataset = result.create_dataset("intensities"
                                        (len(list_of_files), number_of_bins),
                                        dtype='float32')
        for index, one_file in enumerate(list_of_files):
            with fabio.open(one_file) as fimg:
                res = ai.integrate1d(fimg.data, number_of_bins)
        dataset[index] = res.intensity
        result['q'] = res.radial
```

Figure 5
Code snippet for sequential decompression and azimuthal integration.

```python
import h5py, fabio, pyFAI
from threading import Thread, Event
from queue import Queue

class Reader(Thread):
    def __init__(self, queue_in, queue_out, quit_event):
        Thread.__init__(self)
        self._queue_in = queue_in
        self._queue_out = queue_out
        self._quit_event = quit_event

    def run(self):
        while not self._quit_event.is_set():
            idx_fname = self._queue_in.get()
            if idx_fname is None:
                break
            idx, fname = idx_fname
            fimg = fabio.cbfimage.CbfImage()
            fimg = fimg.read(fname, check_MD5=False)
            self._queue_out.put((idx, fimg.data))
            self._queue_in.task_done()
            fimg = None

def process(list_of_files, result_file, geometry_file, number_of_bins, workers=1):
    #prepare the pool of workers:
    queue_in = Queue()
    queue_out = Queue()
    quit_event = Event()
    pool =  []
    for _ in range(workers):
        reader = Reader(queue_in, queue_out, quit_event)
        reader.start()
        pool.append(reader)

    #Send all filename to the queue for processing
    for idx, fname in enumerate(list_of_files):
        queue_in.put((idx, fname))

    #Perform the azimuthal integration and saving
    ai = pyFAI.load(geometry_file)
    with h5py.File(result_file) as result:
        dataset = result.create_dataset("intensities",
                                        (len(list_of_files), number_of_bins),
                                        dtype='float32')
        for _ in list_of_files:
            idx, data = queue_out.get()
            res = ai.integrate1d(data, number_of_bins, method="ocl_csr_gpu")
            dataset[idx] = res.intensity
            queue_out.task_done()
        result['q'] = res.radial

    #Clean up the pool of workers
    queue_in.join()
    queue_out.join()
    quit_event.set()
    for _ in range(workers):
        queue_in.put(None)
```

Figure 6
Code snippet implementing the ‘pool of reader’ strategy and sequential integration.

Acknowledgements

This work has been sponsored by the materials diffraction beamline at ESRF (ID15a). We would like to thank Gavin Vaughan and Marco Di Michiel for providing the test data set composed of 1202 images. We would like to thank V. Armando Solé, head of the Data Analysis Unit and leader of the silx project. Finally, we would like to acknowledge the contribution of all other members of the silx project: Henri Payno, Damien Naudet, Pierre Knobel, Valentin Valls and Pierre Paleo for the great team work.

```python
import h5py, fabio, pyFAI, os
from silx.opencl.codec.byte_offset import ByteOffset

def process(list_of_files, result_file, geometry_file, number_of_bins):
    ai = pyFAI.load(geometry_file)
    data = fabio.open(list_of_files[0]).data
    res = ai.integrate1d(data, number_of_bins, method="ocl_csr_gpu")
    engine = ai.engines["ocl_csr_integr"].engine
    bo = ByteOffset(os.path.getsize(list_of_files[0]), data.size,
                    ctx=engine.ctx)
    cbf = fabio.cbfimage.CbfImage()
    with h5py.File(result_file) as result:
        dataset = result.create_dataset("intensities",
                                        (len(list_of_files), number_of_bins),
                                         dtype='float32')
        result['q'] = res.radial
        for index, one_file in enumerate(list_of_files):
            raw = cbf.read(one_file, only_raw=True)
            dec = bo(raw, as_float=True)
            dataset[index] = engine.integrate(dec)[0]
```

Figure 7
Code snippet for GPU-based decompression and azimuthal integration.

References

Bernstein, H. & Hammersley, A. (2006). *International Tables for Crystallography*, edited by S. Hall & B. McMahon, Vol. G, ch. 2.3, pp. 37–43. New York: Wiley.
Biasci, J. *et al.* (2014). *Synchrotron Radiat. News*, **27**, 8–12.
Blelloch, G. E. (1989). *IEEE Trans. Comput.* **38**, 1526–1538.
Collette, A. (2013). *Python and HDF5.* O'Reilly Media.
Jacques, S. D. M., Di Michiel, M., Beale, A. M., Sochi, T., O'Brien, M. G., Espinosa-Alonso, L., Weckhuysen, B. M. & Barnes, P. (2011). *Angew. Chem. Int. Ed.* **50**, 10148–10152.
Kieffer, J. & Ashiotis, G. (2014). In *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*, Cambridge, UK, 27–30 August 2014, edited by P. de Buyl & N. Varoquaux.
Kieffer, J. & Karkoulis, D. (2013). *J. Phys. Conf. Ser.* **425**, 202012.
Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. & Fasih, A. (2012). *Parallel Comput.* **38**, 157–174.
Knudsen, E. B., Sørensen, H. O., Wright, J. P., Goret, G. & Kieffer, J. (2013). *J. Appl. Cryst.* **46**, 537–539.
Kraft, P., Bergamaschi, A., Broennimann, Ch., Dinapoli, R., Eikenberry, E. F., Henrich, B., Johnson, I., Mozzanica, A., Schlepütz, C. M., Willmott, P. R. & Schmitt, B. (2009). *J. Synchrotron Rad.* **16**, 368–375.
Mokso, R., Schlepütz, C. M., Theidel, G., Billich, H., Schmid, E., Celcer, T., Mikuljan, G., Sala, L., Marone, F., Schlumpf, N. & Stampanoni, M. (2017). *J. Synchrotron Rad.* **24**, 1250–1259.
NIAC (2003). *A common data format for neutron, X-ray and muon science*, http://www.nexusformat.org/.
Rossum, G. van (1989). *Python programming language*, http://www.python.org.
Schmuck, F. B. & Haskin, R. L. (2002). In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, Monterey, CA, USA, 28–30 January 2002, No. 19. USENIX.
Sitaridi, E., Mueller, R., Kaldewey, T., Lohman, G. & Ross, K. A. (2016). *Proceedings of the 45th International Conference on Parallel Processing (ICPP)*, Philadelphia, PA, USA, 16–19 August 2016, pp. 242–247. IEEE.
Sole, V. A., Vincent, T., Kieffer, J., Payno, H., Knobel, P., Naudet, D. & Valls, V. (2015–2017). *silx: collection of Python packages for data analysis at synchrotron radiation facilities*, http://www.silx.org/.
Stone, J. E., Gohara, D. & Shi, G. (2010). *Comput. Sci. Eng.* **12**, 66–72.
Tanaka, H. (2014). *Synchrotron Radiat. News*, **27**, 23–26.
Vamvakeros, A., Jacques, S. D. M., Di Michiel, M., Senecal, P., Middelkoop, V., Cernik, R. J. & Beale, A. M. (2016). *J. Appl. Cryst.* **49**, 485–496.
Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A. & Balakrishnan, V. (2015). *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*, Haifa, Israel, 26–28 May 2015, pp. 6:1–6:11.