Chapter 5

Next-Generation Information Technology Systems for Fast Detectors in Electron Microscopy*

Dieter Weber[†,‡,§], Alexander Clausen[†] and Rafal E. Dunin-Borkowski[†]

[†]*Ernst Ruska-Centre for Microscopy and Spectroscopy with Electrons, Forschungszentrum Jülich GmbH, Jülich, Germany*

[‡]*Peter Grünberg Institute, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany*

[§]*d.weber@fz-juelich.de*

Starting from 2009, the data rate of TEM cameras has outpaced the development of network, mass storage, and memory bandwidth by almost two orders of magnitude. This immense increase in performance opens new doors for applications and, at the same time, requires adequate IT systems to handle acquisition, storage, and processing that go well beyond solutions based on personal computers. This chapter reviews applications, solution strategies, and existing hardware and software components from a practical perspective to scale data handling and processing tasks to match the growing detector performance.

1. Introduction

The Gatan K2 IS direct electron detector,[1] which was introduced in 2014, marked a watershed moment in the development of cameras for transmission electron microscopy (TEM).[2] Its pixel frequency, i.e., the number of data points (pixels) recorded per second, was two orders of magnitude higher than the fastest cameras available only five years before. Starting from 2009, the data rate of TEM cameras has outpaced the development of network, mass storage and memory bandwidth by almost two orders of magnitude (Figures 1 and 2). Consequently, solutions based on personal

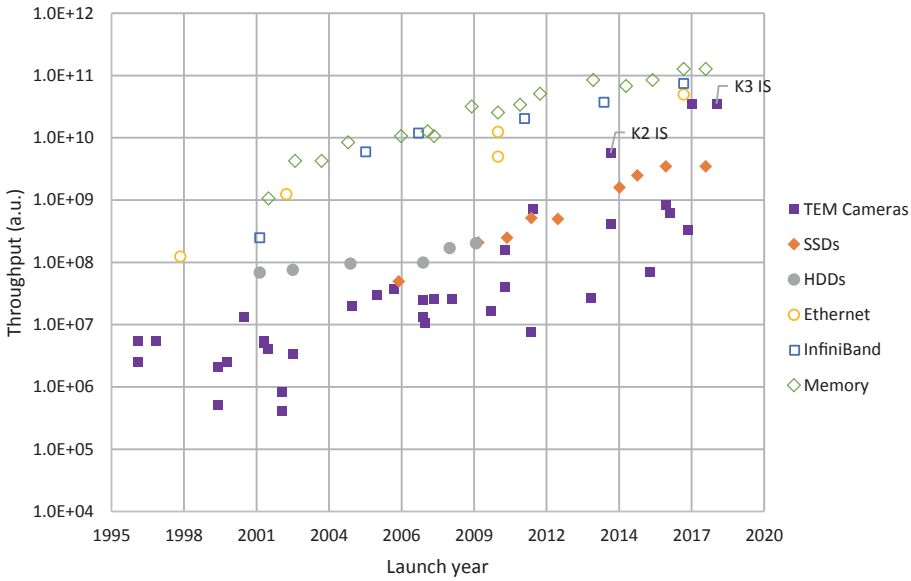

Figure 1. Evolution of TEM camera, solid-state disk (SSD), hard disk drive (HDD), Ethernet, InfiniBand and memory throughput over time. The throughput is given in bytes per second where this information is available. For cameras, the pixel frequency is given in pixels per second. The source file (Ref. 3) is available online and describes in detail how the data was collected and evaluated.

computers (PCs) that were adequate until then are no longer able to handle the resulting data rates. Instead, tailored high-performance set-ups are necessary. Similar developments have occurred for advanced X-ray sources such as the European XFEL, requiring special information technology (IT) systems for data handling.[4,5] Information and detector technology are currently under rapid development and involve disruptive technological innovations. This chapter briefly reviews the technological developments of the past 20 years, presents a snapshot of the current situation at the beginning of 2019 with many practical considerations, and looks forward to future developments.

2. Application of High-Speed Detectors in TEM

In TEM, the first direct electron detectors for recording images were developed primarily to collect data for cryo electron microscopy (cryo EM) in order to increase the detection quantum efficiency by collecting data from individual electrons rather than integrating the dose of many electrons per pixel.[6] Fast detectors with somewhat lower number of pixels and very

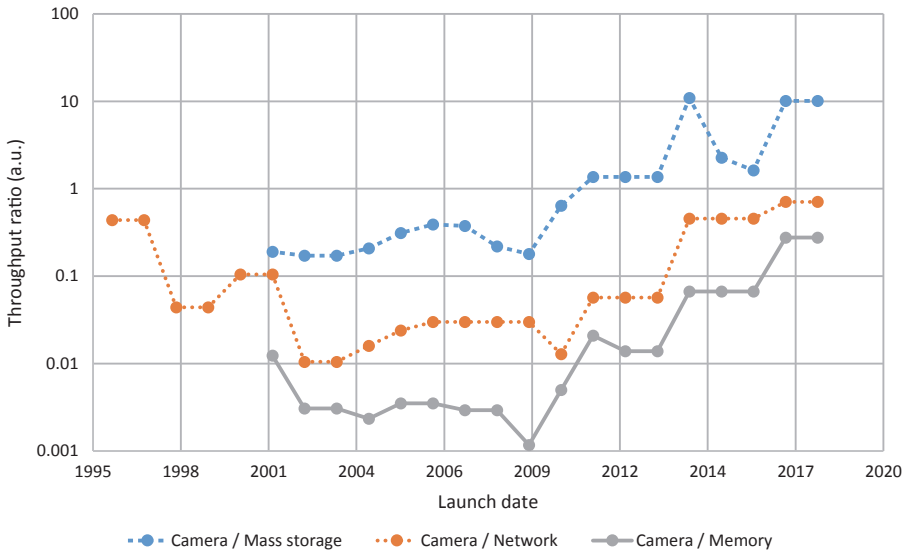

Figure 2. Development of TEM camera pixel frequency relative to typical mass storage, network and memory throughput based on the data in Figure 1.[3] The relative development of camera pixel frequency in comparison to the data rate of mass storage (SSDs and HDDs), Ethernet and memory is highlighted by calculating their quotient. The plot shows that cameras have increased their speed by about two orders of magnitude relative to other IT components since 2009.

high frame rates were developed for pixelated scanning TEM (pixelated STEM),[7–9] where a full electron diffraction pattern is recorded for each position of a STEM scan. It has long been known that collections of electron diffraction patterns, in particular from overlapping sample regions, contain a wealth of information about the sample.[10–15] Such fast detectors provide a practical route to tap into this information for real-world applications,[16–23] because even a small scan of 256 × 256 pixels would take almost two hours to record at 10 frames per second (fps), a speed that was considered state-of-the-art for high-speed TEM cameras until 2010.[3] By using the 400 fps of a Gatan K2 IS, the scan time is reduced to less than three minutes, while the 2400 fps of a Quantum Detectors Merlin camera[24] reduces this time to below 30 s. Other applications of such high-speed cameras include the observation of dynamic processes during *in situ* experiments.[2,25]

3. Challenges and Requirements

Irrespective of their application, using fast electron detectors at their full rates generates large amounts of raw data. This situation creates

significant challenges for data handling and processing.[26,27] One option is to immediately reduce the data in a suitable fashion and to handle only a reduced data stream. This approach is often followed in cryo EM, where frames with individual electron read-outs are combined to some degree. However, users of pixelated STEM and high-speed *in situ* observations benefit immensely from keeping all raw data, so that it can be re-analyzed using different algorithms or settings as desired.[16–19,28] A dataset from a single pixelated STEM scan can already potentially reach a size of up to 64 TB when using a high-resolution (1 k × 1 k) scan with high detector resolution (4 k × 4 k) recording float32 values. Practical pixelated STEM scans acquired with the Gatan K2 IS detector at the Ernst Ruska Centre in Jülich already reach sizes of 200 GB, and a session with several scans can produce 2.5 TB of data. These amounts are expected to grow rapidly as faster cameras, storage and processing systems become available.

In order to make full use of these capabilities, processing and storage have to move to dedicated systems because the data volumes and calculations are beyond what is convenient to handle on individual PCs and hard drives. A first glance at 2.5 TB of data stored on a typical fast PC storage medium using a 300 MB/s read rate would take more than two hours. A time frame of minutes or seconds instead of hours would be desirable for fast turn-around from an experiment to analysis, requiring read rates in the range of 100 GB/s to TB/s.

Furthermore, the acquisition systems themselves have to handle high data rates. For the biggest and fastest detectors, the readout for one detector must be distributed over several PCs, for example by dividing the detector into tiles with parallel readout.[29] Such a high aggregate throughput creates challenges in data transfer from the readout location to a permanent storage, typically requiring a data center close to the acquisition site. Usual PC-based control systems cannot handle such a data rate. Acquisition systems should therefore provide a strongly reduced monitoring signal to the acquisition control interface as a feedback to the user. In view of these requirements, a suitable acquisition system is a complex distributed set-up.

Electron microscopists are used to obtaining immediate visual feedback from the instrument. For pixelated STEM, it is no longer the microscope's electron-optical system or simple electronics that create the image directly on a phosphor screen or monitor, but digital processing of the detector data.[30] Real-time turnaround with processing of results is required to ensure fast analysis and optimal use of instrument time. Since the throughput of a typical computer is fundamentally limited by its memory bandwidth —

currently between 40 GB/s[31] and 160 GB/s[32] — processing has to be distributed across between 10 and 1000 nodes even when using the fastest algorithms to achieve the TB/s data rates that are required for immediate feedback.[33]

Given such requirements, performance and scalability are becoming key enablers for dealing with data. Speed-up results in cost savings or a significant increase in capabilities, for example by being able to use larger data volumes or more complex computations.[34] In contrast to many previous analysis tasks, for which suitable response times could be achieved without special effort, this situation makes investments in performance optimization worthwhile.[35]

Many previous supercomputing applications have involved running complex iterative simulations, such as weather models,[36] fluid dynamics calculations[37] and molecular models[38] very quickly. The emerging data analysis tasks require high throughput,[39] running repeatedly from top to bottom through a very large unchanging or gradually growing dataset and extracting meaningful information for interpretation.

The growing application of supercomputing for data analysis is not limited to fast detectors. It emerges in places where large volumes of data are collected, including the analysis of log files from web services,[40] large collections of high-resolution images or videos,[41] high-density sensor data,[42] and genomics.[43] Many of these tasks are data-parallel, meaning that subsets of the input data can be analyzed in parallel, reduced to a much smaller partial output, and the small partial outputs then combined to produce a complete result. This approach is known as the the "map-reduce" programming model.[44] In general, meaningful reduction or filtering is always necessary to make large data volumes accessible for human beings.

Well-known frameworks that are based on the map-reduce design principle, such as Apache Hadoop and Apache Spark (Mavridis & Karatza, 2017), were originally established to analyze tabular data like log files and databases. The use of such approaches to handle n-dimensional binary data arrays such as high-resolution detector data is emerging for larger sizes of such datasets.[45]

In contrast to traditional high-performance computing or big data analytics, a data analysis system for interactive microscopy should handle live data while it is arriving from the detector and show a preprocessed monitoring result in real time, while at the same time saving raw data to storage. Furthermore, users often require interactive analysis and exploration with immediate *visual* feedback, ideally while a dataset is still being

acquired. Just as for previous smaller datasets, traditional file workflows such as saving, opening, browsing and copying should work in a familiar way, the only difference being the need to handle TBs on a remote distributed system instead of MBs on a local PC. Such systems must therefore be optimized more for low-latency high throughput than for full utilization of all computing resources as in traditional high-performance computing.[46]

4. Solution Strategies

The computer scientist Donald E. Knuth is famously quoted for the phrase "premature optimization is the root of all evil".[47] This phrase is often used to justify building a simple solution first and then optimizing it later. However, throughput in the TB/s region is out of the reach of an ordinary single PC. A system capable of such performance levels is difficult to develop incrementally from a simple PC-based solution and should therefore be designed from the start as a distributed, scalable high-performance solution. Such a system is inherently not the simplest one and it requires substantial optimization. This statement does not contradict Knuth, because his famous quote must be understood in its full context[47]:

> Experience has shown ... that most of the running time in non-IO-bound programs is concentrated in about 3% of the source text. We often see a short inner loop whose speed governs the overall program speed to a remarkable degree; speeding up the inner loop by 10% speeds up everything by almost 10%. And if the inner loop has 10 instructions, a moment's thought will usually cut it to 9 or fewer.
>
> In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering.
>
> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

In order to achieve optimal performance, two factors can be maximized: A system should scale on many parallel worker units that perform the processing, and each of these workers should reach a high performance. The architecture should be validated and its critical code portions should already be identified and optimized in the system design phase, for example using prototypes. The following sections describe various aspects and design considerations for such a system.

5. Optimization of Processing Performance

The throughput per worker can be optimized by identifying and resolving bottlenecks. Profiling yields detailed information on the amount of time the system spends in specific code portions and operating system functions. Based on this information, performance optimizations can be targeted at these critical parts of the system.[47] Array programming,[48] vectorizing operations,[49] eliminating unnecessary copies in the flow of data[50,51] and improving cache efficiency[52] are common strategies that can be used to yield substantial improvements, especially when they are combined.

A scripting language such as Python is usually orders of magnitude slower than a compiled language, and using it for a high-performance application may seem counterintuitive. However, most of the code in a piece of software is not performance-critical[47] and should be written in a language that has a low development and maintenance cost. Furthermore, most applications today use components of pre-existing third-party code and require a high-level glue language to connect them. Python is currently widely used for this purpose in scientific software because it combines easy development with excellent input-output (IO) and interfacing capabilities. Furthermore, it can achieve good numerical performance through suitable extensions. Numpy, for example, efficiently delegates standard numerical tasks to high-performance numerics libraries such as BLAS[53] implementations, PyTorch[54] and Intel MKL[55] through array programming. Numba[56] allows to compile selected code portions just-in-time to generate optimized machine code that can nearly reach the performance of equivalent code written in C or Fortran.

If no pre-existing optimized implementations of a processing task are available, then profiles and tools such as the Compiler Explorer[57] can be used to identify and take advantage of possibilities for speed-up.

6. Optimization of IO Performance and Data Layout

Fundamental design decisions hinge on whether a task is fundamentally CPU-bound or limited by IO. This point is not always easy to judge before optimization, because an optimized implementation can be more than 100 times faster than a naïve version,[58] which can make a CPU-bound implementation become IO-bound. The transfer of data between memory and CPU is often the ultimate bottleneck for numerical codes, not the computation itself.[59] Investment in faster processing units or a

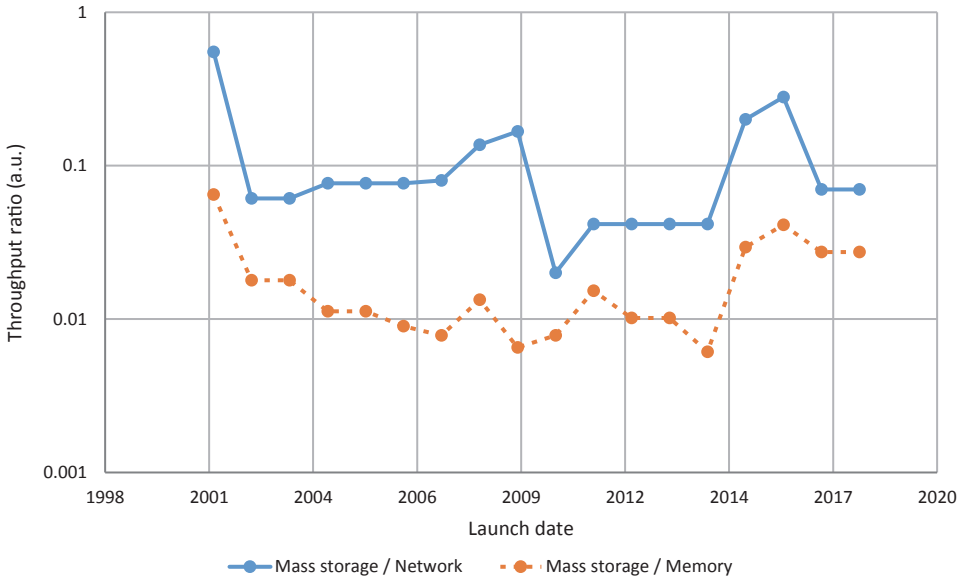

Figure 3. Evolution of mass storage (SSD and HDD) throughput relative to network and memory based on the data in Figure 1.[3] The mass storage throughput is divided by the network and memory throughput to highlight relative changes. It shows a dynamic development with fast network connections emerging between 2009 and 2014, and a dramatic increase of mass storage performance after 2014 driven by innovations in the SSD market. Recently introduced fast networking standards like 100 Gbit Ethernet and 400 Gbit Ethernet may again change the throughput ratio if they find more widespread application.

higher number of cores per system does not necessarily increase performance. Distribution of the processing over many nodes, each of which has high IO and memory bandwidth, but with comparatively low processing speed and good energy-efficiency, can then be a more cost-efficient path.

High IO rates to make data available for processing at worker nodes can be achieved using high-performance network connections to dedicated storage nodes, or by using fast local storage on the worker nodes themselves. Each method can have advantages and disadvantages, depending on the application and available components. As a result of a dynamic development in IT between the years 2000 and 2018, the boundary conditions for designing such systems have changed considerably over time, resulting in a profound impact on the design of high-performance computing infrastructure.

The read rate of traditional hard disk drives (HDDs) fell behind the increase of network performance between 2010 and 2014 (Figure 3). As a result of relatively slow mass storage before 2014 compared to the available

fast network interfaces (Figure 3), it then made sense to build systems that were based on dedicated storage nodes with large disk arrays (RAIDs) of relatively slow magnetic disks with fast network connectivity to worker nodes. This used to be the most attractive solution for high IO performance at computation nodes.[60]

Between 2014 and 2018, flash-memory-based solid-state disks (SSDs) that are directly connected to the ultrafast PCIe buses of modern computers (NVMe SSDs) have provided tremendous progress in performance, while their cost has dropped over time. NVMe SSDs with a capacity of 2 TB that allow reading at around 3.5 GB/s are affordable stock components in 2019. These ultrafast SSDs have closed the gap between high-speed network connections and mass storage, making local storage competitive when compared to network-based solutions.

For all tasks that are data-parallel, i.e., for which the input data can be split into chunks that are stored on separate nodes and processed repeatedly with the map-reduce pattern as described in Section 3, such a set-up can be significantly more cost-effective per aggregate throughput than a network-based solution. It eliminates the need for high-speed storage nodes and only requires commodity network connections such as 10 Gbit Ethernet to initially populate the SSDs, send queries and transfer the results, so long as the results are smaller than the input data.[61] The drawback is that only data-parallel tasks, where each worker reads or writes a small pre-defined part of a dataset, benefit from such an architecture, while a network-based set-up with a suitable cluster file system allows fast random reads and writes over all regions of a dataset.

A cluster that is optimized with fast local storage, but has slow connections between nodes, will only perform well for workloads that involve fast data-parallel reading, processing and writing, and not for workloads that require frequent high-volume low-latency data exchange between nodes, such as running large-scale iterative simulations or frequently re-shuffling a dataset.

The Hadoop File System (Hadoop FS)[62] was developed specifically to manage distributed local storage on processing nodes. It provides an interface of one coherent file system over all storage nodes, while managing splitting into blocks, transfer and replication in a way that is transparent to the user. In contrast to traditional distributed file systems such as Lustre,[63] Hadoop FS is not POSIX-compatible, that means it does not provide all the functionality of a typical file system that conforms to the POSIX standard. The primary difference is that it is append-only, i.e., it does not allow

random writes to an existing file. Since it is optimized for workloads that stream a large dataset to storage from top to bottom and then read many times from this unchanging source file, this is not an obstacle and removes the synchronization effort that would be required to manage changes in a distributed system. A naïve application can read from the Hadoop FS just as from a cluster file system, accessing all parts of a file. This access method typically requires data transfer between the nodes which provides acceptable performance if the network is fast enough — as for a traditional cluster file system.

An application for data-parallel tasks that is aware of the Hadoop file system, can query it about the partitioning and distribution of blocks, partition the processing in a compatible schema, and match tasks to the nodes that hold the required data on their local storage. Under these circumstances, the worker can perform a so-called "short-circuit read"[64] whereby the application obtains an operating system file handle directly to a data block on the local storage medium. The application can access this block directly through local operating system routines with zero overhead, including memory-mapping, through that handle. In such a set-up, the full data is sent through the network only once for writing to local storage on the nodes. Reading is limited to local transfers from the SSDs and does not strain the network.

Depending on the nature of the processing routines, it can be advantageous to re-order the data in order to bring different parts that have to be processed in combination close together on the storage device. An example comes from the use of tiled detectors. One would most naturally write separate files for each tile to achieve high acquisition throughput. The data for a complete frame is then always split up over separate files that could be stored on separate nodes to achieve the highest possible acquisition throughput. However, if a Fourier transform of an entire detector frames is required, then it can make sense to re-shuffle the data so that complete frames are stored together on the same node, as combining the Fourier transforms of partial frames requires substantial data transfer.[65] If all of the data from one frame are kept on the same node, then the data does not have to be sent across the network.

7. Distribution of Computation

Systems such as Apache Spark,[66] which is written in Scala, as well as the Python-based Dask and Dask.distributed,[67] were developed to make computation on distributed systems easier to manage than using the message

passing interface (MPI). Just as the Hadoop file system manages the distribution of storage blocks, Spark and Dask manage the execution flow of computation by processing and combining smaller portions as needed. Spark is optimized to work with distributed storage on the Hadoop file system which is developed specifically for such tasks. It is applied mostly to tabular data, with few provisions for large-scale numerical calculations at the time of writing. Dask handles both tabular and numerical data and has access to the PyData ecosystem, a rich interoperable collection of open source Python modules for a wide range of numerical processing and visualization tasks. Unfortunately, it has no native support to reap the benefits of distributed local storage following the principle of the Hadoop file system at the time of writing.

8. Graphical User Interface

In contrast to typical graphical user interface (GUI) applications, if data processing does not take place on the same PC but on a distributed system, then the front-end and back-end can be far apart, with potentially large round-trip times, over unreliable connections, and with limited throughput between the front-end and back-end. Operations that would finish instantaneously and reliably within a traditional GUI application that runs on a single PC may then take much longer or fail when using such a distributed set-up, for example by losing network connectivity between the front-end and back-end. This situation makes asynchronous operation critical and introduces many more transition and error states that need to be handled.

The development of such a hybrid solution for an interactive GUI with a distributed high-performance data handling and processing back-end requires a suitable system architecture and design approach,[68–72] which is different from previously known batch processing systems that are used in large-scale scientific computing. Large social networks such as Facebook arguable come closest to fulfilling similar requirements,[73] as they have a remote distributed back-end, deal with large-scale data and have a responsive local GUI. On the scientific side, IT systems of the XFEL[5] or systems such as the cesium platform for time-series inference[74] fit into this category.

The React framework for web applications[75] provides a rich ecosystem for user interface components, in combination with state management through libraries such as Redux[69,76,77] and Flux.[70] These frameworks were developed specifically to manage the complex state and asynchronous operation of GUIs for distributed systems. A web-based GUI allows a system to be accessed using standard web technology through any device that can run a modern

web browser, making it a simple and universal choice to develop a responsive GUI for distributed data analysis tasks. Using traditional GUI frameworks is also possible, but requires more work to enable asynchronous operation and to manage the state.

9. IT Infrastructure Management

The development and deployment of a distributed system requires coordination of the operation of many separate computers, including installation, deployment and network configuration. Furthermore, failures of parts of the system occur more frequently when there are more components and higher complexity.

Container management solutions such as Docker[78] and Kubernetes[79] can help to orchestrate the many components of a distributed system, including deployment on commercial or scientific cloud infrastructure.

10. People

Programming languages and software libraries have become so easy to use that basic programming skills are often sufficient to implement effective data analysis workflows and test novel processing schemes with acceptable performance[80] for data volumes and computations that are compatible with PCs.

In contrast, the assembly of existing infrastructure into an effective and user-friendly distributed high-performance processing system for large volumes of data requires a deep understanding of software design and IT systems, from CPU and GPU architecture and instruction sets, through memory and IO handling on the hardware and operating system level, to network architecture, network protocols and web development for remote GUIs.

For this reason, more people who have a primary background in computation and system design are required in this emerging field.[81] Furthermore, data analysis is becoming an interdisciplinary topic between computer science, image processing and application science. In this environment, the development of innovative high-performance IT systems is becoming a key enabler to support progress in the scientific analysis and exploitation of such large-scale data.

11. File Format Requirements

Advanced acquisition and processing tools allow users to generate more data and more complex datasets, and to perform more complex analysis. Furthermore, the movement towards Open Data and Open Science means that users not only exchange analysis results but also publish original data, together with corresponding analysis tools.

For this reason, a standardized open format with rich and reliable metadata that give meaning and context to a dataset and makes it interoperable, discoverable and searchable, is required.[82] Raw data, i.e., a huge collection of data values, is only interpretable in combination with additional information from metadata. Convenient analysis tools should be able to display and meaningfully interpret metadata, ranging from basic information such as the number format and array layout to the instrument settings and sample condition during acquisition.

The file format for detector data should be carefully designed. For raw data, it should allow very efficient, distributed, parallel reading and writing from and to distributed storage, such as the Hadoop file system. For correct and easy interpretation, it should include metadata that follows a standardized schema.

At the time of writing, there is no single catch-all solution. The HDF5 format[83] provides a container that fulfills the requirements with regards to metadata, but it is unfortunately, in its basic Open Source version, incompatible with the way in which systems like the Hadoop FS reach high aggregate IO performance on distributed storage. Furthermore, HDF5 is currently in the process of moving to an "open core" business model, where advanced features, such as support for Hadoop FS, are Closed Source,[84] making it unattractive as a dependency for Open Source high-performance solutions. The use of raw binary data blocks that can be distributed across a distributed storage solution and accessed with the full range of optimized file reading methods that the operating system provides on processing nodes, such as memory-mapping or direct IO, in combination with a sidecar file that contains metadata in a standardized format, currently provides a workable solution that achieves very good IO performance on distributed storage. Such a data layout can be created from a source file through an ingest step or potentially using a transparent caching solution.

12. Open Source Requirement

The publication of such a system under an Open Source license is the only reasonable licensing model.[85]

First, it is scientific software, which means that data processing should be transparent for review. The source code is a direct human-readable description of what a given software does. Scientists as users should be able to modify the software in order to perform new types of analysis. Since a future analysis method can be fundamentally new and different from anything anticipated before, every aspect of the software should be modifiable so that it can be adapted and repurposed for new applications, and to take advantage of new developments in high-performance IT infrastructure.

Second, development is often directly or indirectly publicly funded with tax revenues. Consequently, the results should be available to all stakeholders under fair, reasonable, and non-discriminatory (FRAND) terms, just as the publication of scientific results. Open Source licensing is a straightforward and widespread licensing model to satisfy this requirement.

Third, high-performance software is recompiled often in order to reach optimal performance on a given set-up, requiring access to the source code.

Fourth, installation and cluster deployment are easiest in the framework of an Open Source licensing model. License and compliance management are then straightforward both for developers and for users of such distributed software. There are no technical and very few legal prerequisites to deploy and use the software in any desired form. This situation is very much the opposite when using Closed Source software.

Fifth, Open Source code is advantageous as a foundation for other people's work, for example for integration into third-party software. The use of an Open Source licensing model means that users can rely on the solution remaining available and serviceable under predictable terms and conditions. Even if the original developers abandon the software, the license allows third parties to step in and continue maintenance and development.

Sustaining such a project requires other contributions than income from license sales.[86] Most notably, an Open Source project benefits from in-kind contributions of code, suggestions for improvement, documentation, marketing and support, thereby blurring the lines between developers, managers and users. Other sources of support can be public funding if the software or specific developments are in the general interest of the funding body or enable a project. Support through foundations such as NumFOCUS,[87] the Apache foundation[88] or the Linux foundation[89] plays a big role for

key infrastructure projects. Companies or institutes that are heavy users of a software package can employ key personnel that dedicate most of their time to its development and benefit directly from improvements, maintenance and support. Technology companies like Microsoft,[90] Google[91] and Facebook[92] even develop many in-house projects that primarily serve their own business interests openly as Open Source in order to benefit from additional contributions from third parties and to improve the general user and developer experience for their customers at no cost. Further sources of income can be donations, and service contracts for support and maintenance.

13. LiberTEM as an Example

LiberTEM[93] is an Open Source platform for high-throughput distributed processing of pixelated STEM[94] data, which is being developed in response to challenges and opportunities in handling big datasets from fast detectors for electron microscopy. This section describes the design considerations that guided its initial development in 2018, against a backdrop of existing technology and design principles.

The primary focus of the LiberTEM platform is currently the handling and analysis of pixelated STEM data. The most basic operation for this application is to emulate traditional STEM detectors, which integrate part of the scattered intensity. The pixels of each detector frame are multiplied element-wise with a set of masks and the products for each mask are added up for this type of analysis. A simple binary mask allows the result from any standard STEM detector to be replicated and virtual dark-field imaging to be performed. More complex combinations of masks that have gradients and negative values allow, for example, center of mass calculations and background subtraction from diffraction peaks.

Advanced processing beyond the use of masks can involve any form of image analysis and mathematical processing. Pixelated STEM allows the re-analysis of data using different processing methods on a computer, instead of re-acquiring data at the instrument using different settings, and it allows analysis schemes to be implemented that would be difficult or impossible to achieve with physical systems.

The basic mathematical operation of applying a stack of masks can be simplified by flattening (vectorizing) the scan dimension of a dataset into a linear list. The usually rectangular detector and mask data can be flattened to vectors. The operation becomes a matrix–matrix multiplication of a frame stack with a mask stack, which is one of the most optimized and well-known

data processing tasks.[95] At the same time, it is "embarrassingly parallel". All frames and masks are independent, all element-wise multiplications of a detector pixel with a given mask element are independent, and the summation of these products is associative. The operations can therefore be chunked in nearly any way, the result can be easily assembled from partial results in different ways, and the result is very compact when compared to the input data, since the number of masks is usually low in comparison to the number of detector pixels and frames. It is the ideal situation for processing the data using the map-reduce scheme.

Since modern CPUs and GPUs can perform matrix–matrix multiplications very quickly, the primary task of the LiberTEM platform is to distribute the IO and to keep many cores fed with data, while providing an easy-to-use interface, both in terms of GUI and scripting. No single distributed processing framework currently fulfills all these requirements for interactive pixelated STEM data processing.[96] For this reason, LiberTEM is assembled from several different components.

In order to make the transition from processing on PCs to distributed systems easier, LiberTEM is designed to be easy to install and run on any PC. Unless it is configured otherwise, running it using default parameters starts a local pool of workers, with all components on the same computer. The ambition and value proposition of LiberTEM is to match or exceed the single-node performance and usability of any other processing solution for the supported applications, making it valuable not only as a cluster solution but also as a single node system.

14. Spark Compared to Dask.Distributed

Both Apache Spark and Dask.distributed are designed to make distributed computation easier. In principle they perform very similar tasks,[97] as described below.

On a fundamental level, Spark comprises a well-designed combination of distributed data type, language, execution engine and compiler, with excellent support for data locality on the Hadoop FS. It is written in Scala, which is a functional language that is compiled to run on the Java Virtual Machine (JVM). The primary bottleneck of Spark, when the fundamental design decisions for LiberTEM were made at the beginning of 2018, was the IO performance of the JVM for numerical. When Spark reads binary numerical data from disk and passes it on *via* the Java Native Interface (JNI) to external libraries like a BLAS implementation, the JVM first reads into

an input buffer, copies the data from the input buffer into an internal buffer, and then copies it again into another buffer, which it then passes *via* pointer to the external library. This process limited throughput on a test system to 2.7 GB/s for cached file system reads and typical LiberTEM processing routines. Furthermore, the ecosystem for scientific numerical processing in Spark is much less developed than the Python world at the time of writing.

In contrast, memory mapping a file or letting the operating system copy data into a buffer to pass it on to an external library works with very little overhead in Python. The same task on the same system reaches 14 GB/s, which is more than five times the performance of Spark, with a system implemented in Python that uses Dask.distributed. For this reason, this framework was chosen instead of Spark to distribute tasks on workers.

Unfortunately, Dask.distributed had no native support to exploit data locality when reading from the Hadoop FS at the time of writing. We implemented chunked processing with so-called Dask futures. Such a future defines a processing task, in the context of LiberTEM typically processing a specific part of a dataset, in the form of a function call that the Dask scheduler dispatches for asynchronous execution on one of the workers. The Dask scheduler later makes the result of this function call available to the process that issued the future. This allows us to implement support for data locality ourselves by issuing futures that are assigned to specific workers based on the data layout, and gives us more direct control about the execution process for the live updating of results and for the general subdivision of processing jobs. Furthermore, we can handle live data streams by adding processing jobs for chunks of data as they are coming in.

Dask.array, an emulation of a global data array that is distributed over the processing nodes, is a popular interface to define distributed numerical processing workflows with Dask. Processing tasks are defined in terms of operations on this global array, and Dask computes a task graph to perform the processing task with smaller step-wise calculations on partial data in a distributed and parallel fashion. The task graph is pre-calculated before computation starts and can reach a significant size if a large dataset is processed in small chunks. Furthermore, the task graph calculated by Dask is not always the optimal solution for a given processing task, and it can be difficult to understand and influence how a specific processing task is translated into a task graph. Calculating such task graphs in an optimal fashion for distributed systems that perform non-trivial operations, taking into account data locality and efficiency on all levels, is a hard problem and currently the subject of advanced compiler development.[98]

The flexibility and simplification of calculating task graphs for arbitrary operations is not required in our case since we perform highly optimized recurrent tasks, for which the sequence of processing steps can be hard-coded as the execution flow of our software. With Dask futures, we work on small chunks with an explicitly defined processing sequence that optimizes efficiency on all levels, without taking up memory and computation time for a fine-grained complete task graph.

Benchmarking results for LiberTEM are presented below in Section 21.

15. Architecture

The LiberTEM architecture, which is shown in Figure 4, is built to support high-throughput map-reduce-style computations on big data in a streaming fashion.

The high-level part of LiberTEM that runs on the control node deals with job handling, tasks, division of data and work, and scheduling. This part does not touch the source data, but only handles abstract descriptions of the data, as well as the processing results that are reduced in size. It consists of a client GUI running in the user's browser, which communicates with a web application programming interface (API) *via* HTTP and WebSockets. The web API connects to a Dask.distributed scheduler, which is then responsible for distributing the work onto worker processes and across machines.

As an alternative to the GUI, the user can connect directly to the Dask.distributed cluster *via* the Python API, for example from a Jupyter notebook, from software written in Python, and from software that includes an embedded Python interpreter. An API for other programming languages is currently in planning.

The low-level part of the architecture handles the source data. It consists of worker processes that are distributed on all machines. The worker processes execute tasks, which operate on partitions of the source data. The partitions are statically assigned to nodes, for example using a Hadoop FS namenode. The partitions can be logical blocks of one or several larger files, or they can correspond to individual files of a multi-file dataset. Partitions that are composed of several smaller files are also possible.

The system operates in an asynchronous fashion for progressively updating processing results on the user side. The Python API can wrap this into a batch-oriented synchronous workflow, for example for scripting.

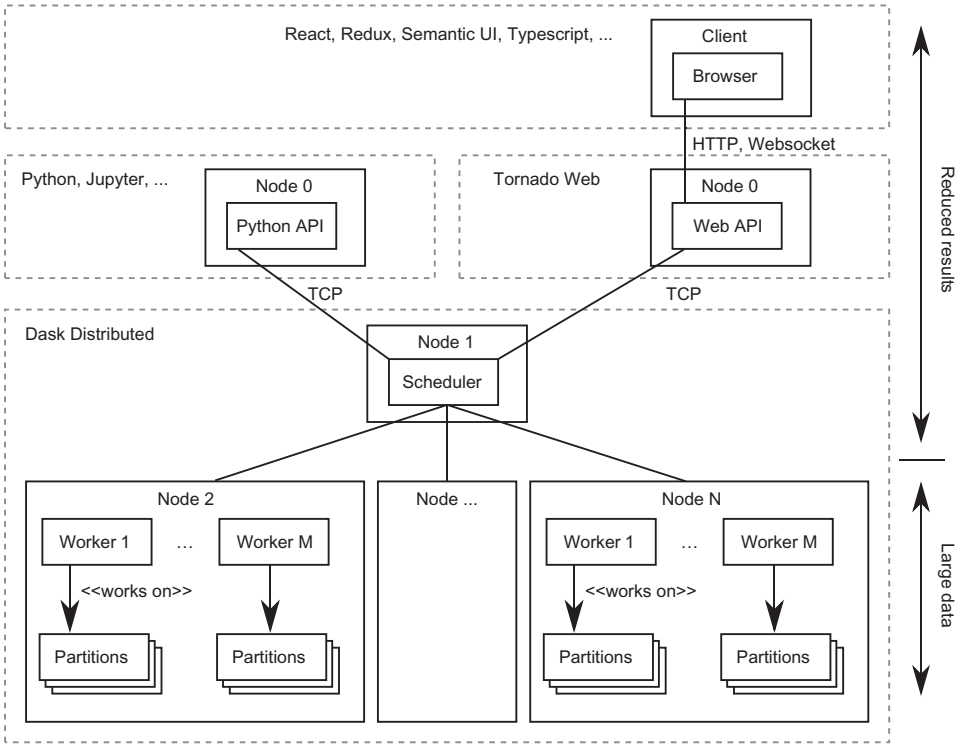

Figure 4. Architecture of LiberTEM to perform map-reduce operations on large-scale binary datasets with high aggregate throughput. A dataset split into partitions that are assigned to the workers on the processing nodes. That allows to store the partitions on local storage for fast and scalable read access. The web API or Python API sends processing tasks that operate on these partitions to the scheduler, which then dispatches them to the workers and feeds the results back to the API. There, the partial results are assembled and returned to the user. This process can be performed asynchronously and progressively to achieve live updating results on the user side.

16. Execution and Data Flow

In order to start processing a dataset, the user creates a job, which is an abstract description of the operation to perform, including a reference to a dataset with the parameters for the operation, for examples the masks to apply.

The user can then instruct LiberTEM to run the job. The dataset is subdivided into partitions, whose size is chosen in such a way that their processing finishes in about 100 ms to achieve a real-time feeling for the user.[33] A size of 256 MB to 512 MB achieves this response time for typical mask application tasks when reading from the file system cache. Smaller

partitions create more overhead for scheduling, while larger partitions result in less responsive feedback.

For each partition, a task is created, which is scheduled by Dask.distributed. It is executed on a worker that has fast access to the partition, i.e., one running on a node that keeps the partition on its local storage. The high-level task handling part of LiberTEM can query a Hadoop FS name node to list the nodes that hold a given partition. Details about task execution within Dask.distributed are given elsewhere.[99]

When a task is executed on a worker process, it reads source data from its associated partition. This is done preferentially by memory mapping (mmap) the entire dataset or partition and delegating cache-efficient access to the underlying computation if the node has enough memory to keep all source data in its file system cache. For formats that require more pre-processing or decoding before computation, or under memory pressure, the partition is read in small tiles that fit comfortably into the L3 cache for each worker process. We currently run one worker process per CPU core that processes a single tile at a time. Tile sizes of approximately 1 MB give good results for the typical L3 cache sizes of current CPUs.

After running the computation, the reduced result is sent back to the user's machine and combined with the results of the other tasks to build up the complete result.

When using the web API and web-based GUI, the job is created when the user clicks the "Apply" button. As a job can take some time to complete and the web API may need to respond to other requests in the meantime, this part of the code is written using the Asyncio package and async/await keywords that were introduced in Python 3.5. In this way, the event loop of the web API is never blocked during long-running operations and remains responsive.

17. GUI and API

Figure 5 shows a screenshot of the web application GUI of LiberTEM. Most user interaction, such as changing the size and position of the mask or adding a new analysis, runs locally in the browser. Clicking "Apply" starts a calculation on the back-end, i.e., creates a job and sends processing tasks to the workers. The GUI remains responsive while an analysis is running and shows partial results as they arrive from the processing back-end. The GUI allows several windows, files and analyses, which can operate in parallel, to be opened.

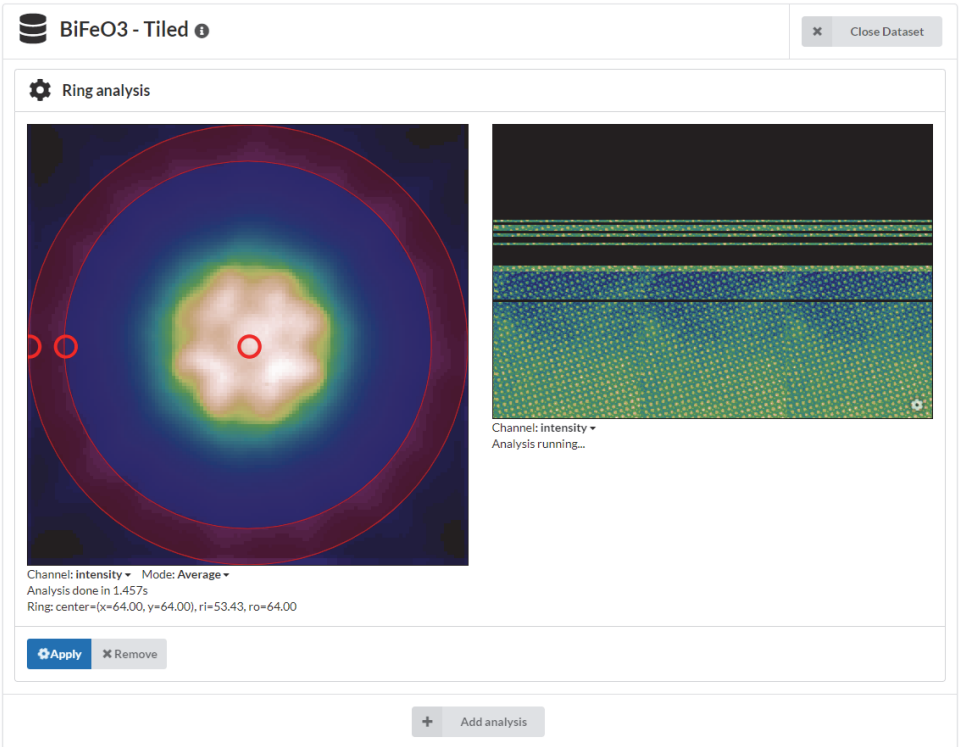

Figure 5. GUI of LiberTEM while processing a virtual high-angle annular dark-field image on a 24 GiB semi-synthetic 4D STEM dataset. On the left, the GUI can show an average of all detector frames or individual frames. The user can define the size and position of the ring that is used as a mask for integrating the intensity on all detector frames. On the right, the processing result is displayed in the form of a virtual STEM image. The GUI shows partial results while the calculation is progressing. Each stripe corresponds to an individual partition. Black stripes indicate results that are still pending.

Figure 6 shows an example of using LiberTEM through its scripting API in a Jupyter notebook. The asynchronous back-end is encapsulated with a synchronous wrapper to make it easy to use in such a scripting environment. The asynchronous API can be used for integration in a third-party application with the same responsive and interactive behavior as the web GUI.

18. Numerical Processing

Since LiberTEM implements basic pixelated STEM processing using matrix multiplication, it takes advantage of existing high-performance software libraries for this purpose. Usually, only a few masks are applied per

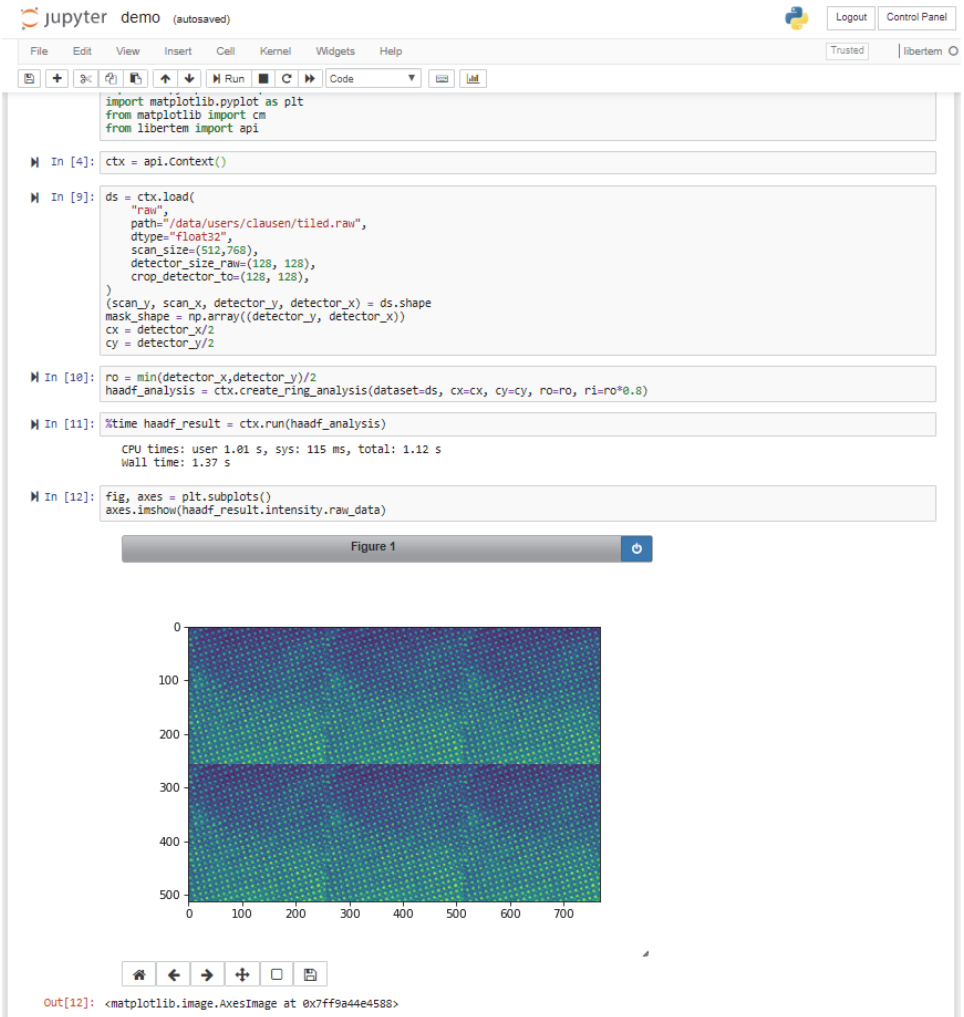

Figure 6. Example of using LiberTEM within a Jupyter notebook. This script calculates a similar result as in Figure 5 using the Python API of LiberTEM. Details regarding the API and further examples can be found online as part of the LiberTEM documentation.

processing job, meaning that the mask stack matrix is tall and skinny. The dataset can be non-contiguous when parts of it, such as frame headers, are masked out. Under these circumstances PyTorch currently achieves the best performance, up to two times faster than Numpy using Intel MKL BLAS or OpenBLAS. As a fallback, LiberTEM can use the BLAS implementation that is accessible through the Numpy Python package on the system.

More complicated processing schemes may require the development of efficient kernels, as a Python and Numpy implementation typically cannot

use all possibilities for optimization, like variable elimination and optimal use of CPU caches and registers. Bit packing and unpacking is an example for a processing task that is difficult to implement efficiently using Numpy. Such kernels are often written in C, C++ or Fortran, because these languages traditionally have the best compilers and tools to transform the source code into highly efficient machine instructions.

Since only a very small percentage of the source code, perhaps even only a few lines, is actually performance-critical,[47] the use of a different programming language with its own full-blown build chain just for these lines of code is a disproportionate effort. Furthermore, compiling the code to use optimal machine instructions for any given machine can create significant efforts and introduce error sources both for development and deployment of a software project.

Numba, a package that compiles Python code of selected functions just-in-time (JIT) to highly optimized machine instructions for the native CPU or GPU architecture through the LLVM compiler infrastructure,[100] has turned out to be very promising. When compared to an optimized C, C++ or Fortran implementation, Numba source code is much easier to manage and deploy on a given machine. Numba manages a self-contained small build chain that is very easy to deploy on a target system simply by installing the Numba Python package. When combined with the flexibility of Python, it even allows new versions of a function to be compiled on-demand, which can be advantageous under some circumstances.

For numerical processing, it reaches nearly the same performance as highly optimized C or C++ code, because it can use the capabilities of the LLVM stack, most notably variable elimination and auto-vectorization to use "single instruction, multiple data" (SIMD) instruction sets. Furthermore, it allows routines that process the data in fine-grained blocks to be written, which can benefit from the L1 and L2 CPU caches and would incur significant overheads for Python with Numpy.

As an example, LiberTEM uses a Numba implementation of a fast vectorized decoder for packed uint12 data (Figure 7) as part of a high-throughput file reader that reaches 85% of the performance of an equivalent C version. Although this may sound trivial, bit packing and unpacking with optimal throughput using SIMD instructions can be a challenging task because of data alignment issues. Only modern compilers can generate efficient auto-vectorized machine instructions for this task from simple source code. Hand-optimized intrinsics with very significant development effort had to be used previously.[101] In contrast, the implementation in LiberTEM is *both* simple *and* fast.

```python
@numba.njit
def decode_uint12_le(inp, out):
    """
    decode bytes from bytestring ``inp`` as 12 bit into ``out``

    based partially on https://stackoverflow.com/a/45070947/540644
    """
    assert np.mod(len(inp), 3) == 0

    for i in range(len(inp) // 3):
        fst_uint8 = np.uint16(inp[i * 3])
        mid_uint8 = np.uint16(inp[i * 3 + 1])
        lst_uint8 = np.uint16(inp[i * 3 + 2])

        a = fst_uint8 | (mid_uint8 & 0x0F) << 8
        b = (mid_uint8 & 0xF0) >> 4 | lst_uint8 << 4
        out[i * 2] = a
        out[i * 2 + 1] = b
```

Figure 7. Fast 12 bit decoder written in python and compiled with numba. By using this decoder, liberTEM reaches an aggregate processing throughput of 8 GiB/s on a Xeon W-2195 with 18 cores when applying a single mask operation to K2 IS raw files that consist of packed 12 bit integers, corresponding to 21 GiB/s of float 32 numbers pixel-for-pixel, which is as fast as processing float 32 directly using liberTEM.

19. Processing Unit

In principle, graphics processing units (GPUs) are excellent at performing matrix multiplications. However, GPUs are diverse, not all PCs have one that can outperform its central processing unit (CPU) for mathematical tasks, and not all GPUs have fast enough memory transfer to match the speed of the CPU in throughput-limited tasks. For these reasons, LiberTEM currently implements a fast CPU version of all of its capabilities. The use of available GPUs as "boosters", with both CPU and GPU workers in parallel, is in planning at the time of writing.

20. Support for the Hadoop FS

Since LiberTEM should run on any PC as a local system, it can read from any file system. For cluster deployment, a system like the Hadoop file system is ideally suited for a massively data-parallel task like pixelated STEM data processing. For this reason, LiberTEM is designed to take maximum advantage of such a set-up if it is available. Not all file formats, most notably HDF5, are well suited for distributed storage. LiberTEM therefore

implements an optimized custom data layout for distributed storage that can be created from any suitable input file using an ingest step. The layout consists of a sidecar file with metadata and several raw binary files that constitute the input data partitions used in processing. A distributed file system can then distribute and replicate these separate blocks through the cluster.

The processing follows this partitioning scheme and distributes tasks to the nodes that hold such a partition in their storage. The workers can use the short-circuit read process of the Hadoop FS for blocks that are on the local storage. In this way, LiberTEM achieves high aggregate throughput and excellent scalability.

The "Benchmarking results" that are given in Section 22 were not generated on a Hadoop FS, but simulate such a set-up by copying the entire dataset to each node to simplify the tests.

21. File Format Support

LiberTEM provides users with high throughput, responsiveness and ease of use, in particular for initially exploring a new dataset after acquisition. It allows many file types to be read in order to avoid file conversion where possible. Current support includes raw data such as data from the EMPAD detector,[9] Quantum Detectors MIB, Gatan K2 IS raw data, Nanomegas Block Files, SER files, FRMS6 files for PNDetector, and all usual HDF5-based formats, such as NeXus, Hyperspy and EMD files.

For read performance, the ideal format is raw data in one of the standard number formats supported on the CPU architecture and the underlying BLAS library, most notably float32 in row order (C order), i.e., detector frame after detector frame without any frame headers or other gaps. Such a dataset can be memory-mapped and passed directly to numerics libraries such as PyTorch or BLAS. Interaction between the numerics library's memory access pattern and the operating system's paging mechanisms for memory mapping provides optimal performance if enough free memory is available. If the file is already in the file system cache, then accessing a memory-mapped file on Linux is as fast as accessing a buffer in memory.

File formats that store packed integers, like uint12, are currently able to reach approximately the same performance level number-for-number. The processing time for the additional decoding step and transfer into a small extra buffer is compensated by a much smaller size on the storage medium, which improves the read rate as well as the memory use of the file system

cache. Working on buffers that fit comfortably into the L3 CPU cache means that the subsequent processing step can access the buffer from the L3 cache instead of the main memory.

HDF5 achieves good performance on single-node set-ups. It can only be memory-mapped for the special case of contiguous data without chunking, which means that it nearly always incurs overheads from copying and HDF5-internal processing. On a Hadoop file system, it unfortunately neither allows distribution onto separate storage blocks nor high-performance reading with a short-circuit read with its native functions. It can therefore currently only benefit from the advantages of the Hadoop FS for data-parallel processing if the data is saved in contiguous mode and the payload data is accessed directly in raw binary form, bypassing the entire HDF5 software stack for the read operations.

In order to manage the metadata, in contrast to numerical data, the HDF5-based NeXus format[102] for neutron, electron and X-ray datasets is currently the most promising solution in the context of LiberTEM. Metadata support and a format definition for pixelated STEM data are under development at the time of writing.

The advantage of NeXus is not so much the encoding of the data, but a strict, well-defined and managed namespace for metadata, including application-specific schema definitions and automated validation tools. This makes the metadata unambiguous and suitable for automated processing across different applications. In principle, the namespace can be encoded in different ways, depending on the application.

22. Benchmarking Results

LiberTEM was benchmarked on a Supermicro MicroCloud 5038MD-H8TRF blade system with the configuration as: illustrated in Table 1.

The raw read performance of the SSD RAID was tested with the "Flexible I/O Tester" (FIO)[103] and reached on average 6711 MiB/s, which corresponds precisely to the specified performance of the used SSDs.

IO-bound operation processing performance with LiberTEM was tested on a semi-synthetic 4D STEM dataset for an array of 10,240 × 768 frames of 128 × 128 pixels each, amounting to 480 GiB of float32 values in form of a raw binary file. The CPU-bound operation, i.e., reading from the file system cache, was tested for a semi-synthetic 512 × 768 array of frames with a size of 128 × 128 pixels, amounting to 24 GiB of float32 values. The files were generated from a smaller real 4D STEM dataset by tiling it in order

Table 1. Description of the test cluster configuration.

Base system	Supermicro MicroCloud 5038MD-H8TRF
File system	XFS
OS	CentOS 7.6
Kernel	3.10.0-957.5.1.el7.x86_64
CPU Cluster nodes	Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
CPU Head node	Intel(R) Xeon(R) W-2195 CPU @ 2.30GHz
Network interface (cluster)	Intel Ethernet Controller 10G X550T
Network interface (Head node)	Intel Ethernet Controller X540-AT2
Switch	ZyXEL XS3700-24
Storage	2x Samsung SSD 970 EVO 2TB on each node
Distributed set-up	Dask scheduler and dask client in version 1.26.0 running on head node, dask worker running on each cluster node,
FIO version	3.1
Python version	3.6.6
PyTorch version	1.0.1.post2
Torch backend	MKL

to reach a sufficient file size. The files were copied to each node for testing. In production, a suitable file system such as the Hadoop File System, or in the future a caching layer, can be used to use the local storage space more efficiently.

A virtual detector operation with a single random mask was performed on the dataset and the time to completion was recorded. The system was using PyTorch with an Intel MKL back-end. The file system caches were dropped for each IO-bound test run and populated for each CPU-bound run.

IO-bound processing throughput with LiberTEM on the cluster scaled nearly linearly with the number of nodes, up to the available maximum number of eight nodes (Figure 8).

The IO-bound processing rate saturated at 5945 MiB/s on a single node, which is approximately 700 MiB/s below the maximum available read rate in FIO. This value was achieved using direct IO which circumvents the file system cache. The use of mmap or conventional file reads achieved lower performance (Figure 9).

CPU-bound processing reached a much higher throughput of 11,407 MiB/s and tapered off only moderately with the number of worker processes until all cores were running a worker process. For comparison, the same CPU-bound test was performed on the high-performance head node with 18 cores (Figure 10).

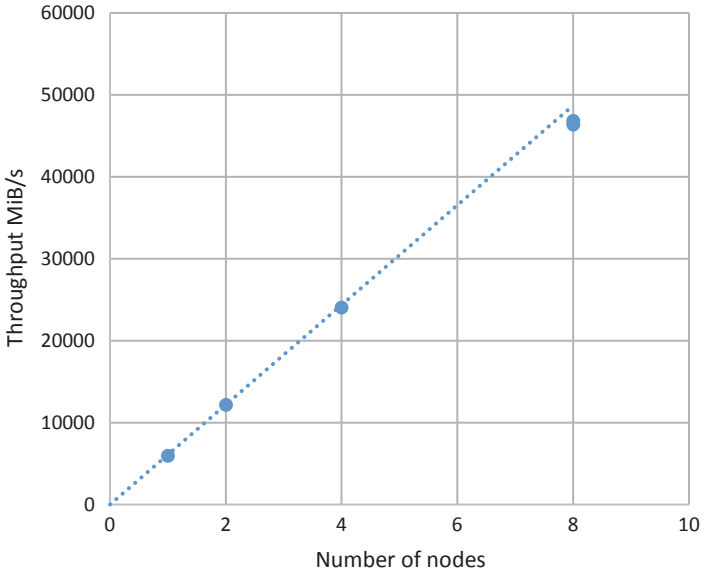

Figure 8. IO-bound scaling behavior of the test cluster described in Table 1 as a function of the number of processing nodes. Each node was running with eight worker processes, i.e., one worker process per real core.

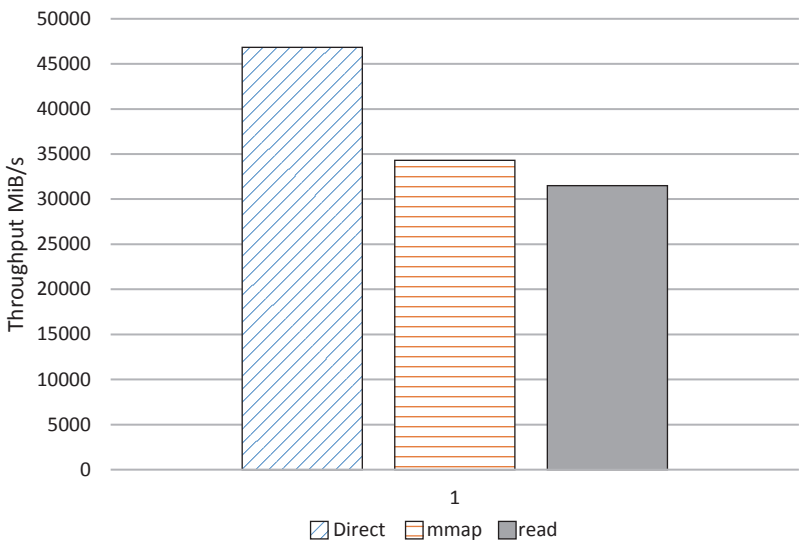

Figure 9. Comparison of aggregate processing rate of the test cluster (Table 1) with reading methods tile-wise direct IO, mmap and conventional tile-wise file reading in an IO-bound scenario under memory pressure, i.e., with a file that is larger than the available memory. Direct IO avoids the overheads associated with constantly moving pages in and out of the operating system's page cache under such conditions.

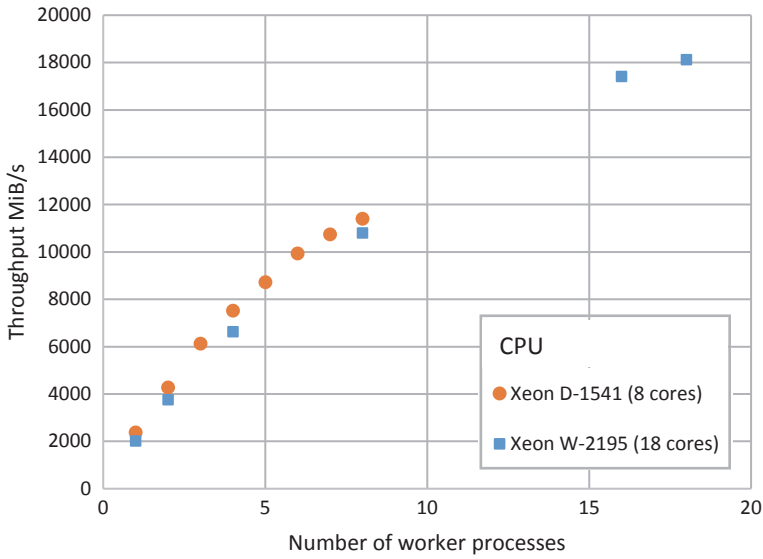

Figure 10. Comparison of CPU-bound scaling behavior on a single processing node and the head node as a function of the number of worker processes. MMAP from the file system cache was used for reading.

The benchmark results show how an application that is designed for data-parallel processing can benefit from the use of local storage to achieve very high aggregate IO rates on a simple and compact system built from stock components as described in Table 1. A map-reduce-like processing scheme allows very fast and scalable applications to be created with a comparatively simple design.

Achieving such performance required a few targeted optimizations, including eliminating copy operations from the data flow as much as possible. For files that did not fit into memory, direct IO that bypasses the file system cache proved to be most efficient by avoiding overheads from constantly thrashing the cache under heavy memory pressure. Comparison with the FIO benchmarks show that the reader can still be improved, possibly by using a multithreaded reader or advanced interfaces such as liburing,[104] which are optimized to read from fast SSDs.

Mmap turned out to be the most efficient method to read from files that can be held in the file system cache. This reading method allows the processing algorithms to operate directly on the file system cache without allocating buffers or making extra copies, if there is enough memory available to hold all of the data.

The linear scaling behavior as a function of the number of processing nodes (Figure 8) shows that the performance can likely be further increased by adding more processing nodes, until the increased load and network traffic on the head node create a bottleneck.

The processing performance on a single node in the CPU-bound scenario scaled well with the number of worker processes up to the number of available physical cores. Hyperthreading provided only a marginal increase. Using a map-reduce-like processing scheme reduces interaction between the workers and makes efficient parallelization easy.

23. Development Roadmap

For the future, a caching layer that uses local SSDs to cache data from remote sources in an optimized layout without needing the Hadoop file system, a data transformation layer and support for vectored IO are planned.

The processing live data streams during acquisition while streaming them to storage, with the possibility to re-process a growing dataset as desired, is in planning as a major update.

Also highly desirable are the application of user-defined functions to a dataset. This feature is nearly completed at the time of writing and will likely soon be included in an upcoming release.

Support for physical units and metadata is in the planning stage.

The embedding of LiberTEM in third-party applications is currently at a prototype stage, with a first demonstration achieved for integration in an alpha version of the upcoming Digital Micrograph release by Gatan that supports Python.

24. Conclusion and Outlook

Building responsive GUI applications for the scalable, distributed processing of very large datasets is a rapidly developing field. It will become a key enabler to work with large datasets that are stored in the cloud, because it allows the processing back-end of such an application to be containerized and executed close to the data storage location, where it can achieve high throughput with little data transfer, while the front-end can even run on a mobile device on a different continent.

The eResearch infrastructure in Australia already implements such workflows, for example coordinated by Microscopy Australia for microscopy applications. Currently, normal desktop applications must be packaged and

deployed as virtual machines using a web-based virtual desktop. This layered workaround highlights the gap in requirements between traditional desktop GUI applications that work on local systems and future needs for cloud-based interactive processing. In the future, we expect that more data processing solutions will be developed natively to reap the benefits of distributed storage and processing, while using web technology for their front-ends so that they are ready for cloud deployment.

For electron microscopy, responsive and interactive processing of live detector data with well-defined open programming interfaces, convenient user interfaces, high throughput and very short response times below 100 ms will be a key towards future applications where automation and algorithms are working hand in hand with human interaction and data interpretation. Dynamic instrument control may even require response loops with only a few milliseconds turn-around.

Detector data with a high information content can be analyzed by using forward models in a feedback loop that compare acquired data from multiple detectors with simulations based on models of the specimen. This approach is already used with very good results to reconstruct the magnetization states of samples.[105,106] The implementation of such an analysis would require a hybrid system that combines fast iterative optimization of a model with the handling of a massive dataset. If it can be scaled up to provide real-time operation, then it can inform a microscope what data to acquire next, in order to gather required information to refine a sample model to a desired level of precision.

In the future, we envision an integrated solution that combines the use of correlative methods, simulation, visualization, instrument control and the full automation of workflows in a suitable system architecture. Such a system can be built of interoperable components from different scientific and commercial contributors. Open standards and open interfaces, in combination with Open Source core components, can enable the development of such an advanced solution.

A heterogeneous system of many components connected using a glue language leaves potential for optimization. For performance-critical parts, a framework that handles distributed computation and data locality (such as Spark) and allows easy development and interfacing (like Python) but is compiled down to machine code and incorporates full-scale integrated optimization, from high-level variable elimination, blocking and data locality to cache-efficient processing and optimal machine instructions, would be desirable.

Improvements in the infrastructure for developing complex high-performance distributed applications are currently tasks for computer scientists and IT specialists. We hope that in the future scientists will again be able to implement their analysis workflows *ad hoc* using simple tools, without a deep knowledge of the underlying mechanisms. Systems such as LLVM Polly,[98] Apache Spark, Dask.distributed and LiberTEM are already working in this direction.

Funding

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 780487). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823717-ESTEEM3.

The Ernst Ruska Centre at the Jülich Research Center supports development through employing key contributors and providing the required infrastructure for development and testing.

Acknowledgments

The authors would like to acknowledge the ESTEEM3 network, many colleagues at Forschungszentrum Jülich and other institutions, and developers and users of various software and hardware solutions for valuable insights and discussions. https://libertem.github.io/LiberTEM/acknowledgments.html.

References

1. Gatan Inc. K2 IS camera. (2018). Retrieved from https://web.archive.org/web/20180809021832/http://www.gatan.com/products/tem-imaging-spectroscopy/k2-camera.
2. Pan, M. & Czarnik, C. Image detectors for environmental transmission electron microscopy (ETEM). In *Controlled Atmosphere Transmission Electron Microscopy* (Springer International Publishing, 2016), pp.143–164, doi: 10.1007/978-3-319-22988-1_5.
3. Weber, D. Development of IT system and TEM camera performance. Zenodo. (2018), doi: 10.5281/zenodo.2450624.
4. Sauter, N. K., Hattne, J., Grosse-Kunstleve, R. W. & Echols, N. New Python-based methods for data processing. *Acta Crystallograp. Sec. D.* **69**, 1274–1282 (2013), doi: 10.1107/s0907444913000863.
5. Fangohr, H. *et al.* Data analysis support in Karabo at European XFEL. In *Proc. 16th Int. Conf. Accelerator and Large Experimental Control Systems, ICALEPCS2017, Barcelona, Spain.* (2018), doi: 10.18429/jacow-icalepcs2017-tucpa01.

6. Li, X. *et al.* Electron counting and beam-induced motion correction enable near-atomic-resolution single-particle cryo-EM. *Nature Methods* **10**, 584–590 (2013), doi: 10.1038/nmeth.2472.
7. Simson, M. *et al.* 4D-STEM imaging with the pnCCD (S)TEM-camera. *Microsc. Microanal.* **21**, 2211–2212 (2015), doi: 10.1017/s1431927615011836.
8. Yang, H. *et al.* 4D STEM: high efficiency phase contrast imaging using a fast pixelated detector. *J. Phys. Conf. Ser.* **644**, 012032 (2015), doi: 10.1088/1742-6596/644/1/012032.
9. Tate, M. W. *et al.* High dynamic range pixel array detector for scanning transmission electron microscopy. *Microsc. Microanal.* **22**, 237–249 (2016), doi: 10.1017/s1431927615015664.
10. Cowley, J. M. Coherent interference in convergent-beam electron diffraction and shadow imaging. *Ultramicroscopy* **4**, 435–449 (1979), doi: 10.1016/s0304-3991(79)80021-2.
11. Hegerl, R. & Hoppe, W. Dynamische Theorie der Kristallstrukturanalyse durch Elektronenbeugung im inhomogenen Primärstrahlwellenfeld. *Ber. Bunsen. phys. Chem.* **74**, 1148–1154 (1970), doi: 10.1002/bbpc.19700741112.
12. Hoppe, W. Beugung im inhomogenen Primärstrahlwellenfeld. I. Prinzip einer phasenmessung von elektronenbeugungsinterferenzen. *Acta Crystallogr. Sec. A* **25**, 495–501 (1969), doi: 10.1107/s0567739469001045.
13. Humphreys, C. J., Eaglesham, D. J., Maher, D. M. & Fraser, H. L. CBED and CBIM from semiconductors and superconductors. *Ultramicroscopy* **26**, 13–23 (1988), doi: 10.1016/0304-3991(88)90371-3.
14. Nellist, P. D., McCallum, B. C. & Rodenburg, J. M. Resolution beyond the information limit in transmission electron microscopy. *Nature* **374**, 630–632 (1995), doi: 10.1038/374630a0.
15. Steeds, J. W. Convergent beam electron diffraction. In *Introduction to Analytical Electron Microscopy* (Springer US, 1979), pp. 387–422, doi: 10.1007/978-1-4757-5581-7_15.
16. Jiang, Y. *et al.* Electron ptychography of 2D materials to deep sub-ångström resolution. *Nature* **559**, 343–349 (2018), doi: 10.1038/s41586-018-0298-5.
17. Krajnak, M., McGrouther, D., Maneuski, D., Shea, V. O. & McVitie, S. Pixelated detectors and improved efficiency for magnetic imaging in STEM differential phase contrast. *Ultramicroscopy* **165**, 42–50 (2016), doi: 10.1016/j.ultramic.2016.03.006.
18. MacLaren, I. *et al.* Pixelated STEM detectors: opportunities and challenges. In *European Microscopy Congr. 2016: Proc.* (American Cancer Society, 2016), pp. 663–664, doi: 10.1002/9783527808465.EMC2016.6284.
19. Yang, H., Pennycook, T. J. & Nellist, P. D. Efficient phase contrast imaging in STEM using a pixelated detector. Part II: optimisation of imaging conditions. *Ultramicroscopy* **151**, 232–239 (2015), doi: 10.1016/j.ultramic.2014.10.013.
20. Nguyen, K. X. *et al.* 4D-STEM for quantitative imaging of magnetic materials with enhanced contrast and resolution. *Microsc. Microanal.* **22**, 1718–1719 (2016), doi: 10.1017/s1431927616009430.
21. Ophus, C., Ercius, P., Sarahan, M., Czarnik, C. & Ciston, J. Recording and using 4D-STEM datasets in materials science. *Microsc. Microanal.* **20**, 62–63 (2014), doi: 10.1017/s1431927614002037.
22. Pennycook, T. J., Lupini, A. R., Yang, H., Murfitt, M. F., Jones, L. & Nellist, P. D. Efficient phase contrast imaging in STEM using a pixelated detector. Part 1: experimental demonstration at atomic resolution. *Ultramicroscopy* **151**, 160–167 (2015), doi: 10.1016/j.ultramic.2014.09.013.

23. Sagawa, R. *et al.* Development of fast pixelated STEM detector and its applications using 4-dimensional dataset. *Microsc. Microanal.* **23**, 52–53 (2017), doi: 10.1017/s1431927617000940.
24. Quantum Detectors. Merlin for EM technical datasheet. (2017). Retrieved from http://quantumdetectors.com/wp-content/uploads/2017/01/1532-Merlin-for-EM-Technical-Datasheet-v2.pdf.
25. Lawrence, E. L., Chang, S. L. & Crozier, P. A. *In situ* TEM observations of oxygen surface dynamics in CeO_2 cubes. *Microsc. Microanal.* **23**, 1994–1995 (2017), doi: 10.1017/s1431927617010637.
26. Belianinov, A. *et al.* Big data and deep data in scanning and electron microscopies: deriving functionality from multidimensional datasets. *Adv. Struct. Chem. Imaging* **1**, (2015), doi: 10.1186/s40679-015-0006-6.
27. Jesse, S., Chi, M., Belianinov, A., Beekman, C., Kalinin, S. V., Borisevich, A. Y. & Lupini, A. R. Big data analytics for scanning transmission electron microscopy ptychography. *Sci. Rep.* **6**, (2016), doi: 10.1038/srep26348.
28. McCallum, B. C. & Rodenburg, J. M. Simultaneous reconstruction of object and aperture functions from multiple far-field intensity measurements. *J. Opt. Soc. Amer. A* **10**, 231 (1993), doi: 10.1364/josaa.10.000231.
29. X-Spectrum. Lambda2M large area Medipix3 based detector array. Retrieved from https://web.archive.org/web/20181008123303/https:/x-spectrum.de/index_htm_files/X-Spectrum_datasheet_2M.pdf.
30. Li, X., Dyck, O., Kalinin, S. V. & Jesse, S. Compressed sensing of scanning transmission electron microscopy (STEM) on non-rectangular scans, preprint (2018). https://arxiv.org/abs/1805.04957.
31. Intel. Intel® Xeon® E-2124 Processor. (2018). Retrieved from https://ark.intel.com/products/134856/Intel-Xeon-E-2124-Processor-8M-Cache-up-to-4-30-GHz-.
32. Wikichip.org. EPYC 7601 - AMD. (2018). Retrieved from https://en.wikichip.org/wiki/amd/epyc/7601.
33. Nielsen, J. *Usability Engineering* (Elsevier Science, 1994). Retrieved from https://www.sciencedirect.com/book/9780125184069/usability-engineering.
34. Delvecchio, P. De-mystifying software performance optimization. (2011). Retrieved from https://software.intel.com/en-us/articles/de-mystifying-software-performance-optimization.
35. Busch, K. The rules of optimization: why so many performance efforts fail. (2016). Retrieved from https://hackernoon.com/the-rules-of-optimization-why-so-many-performance-efforts-fail-cf06aad89099.
36. Bauer, P., Thorpe, A. & Brunet, G. The quiet revolution of numerical weather prediction. *Nature* **525**, 47–55 (2015), doi: 10.1038/nature14956.
37. Tezduyar, T., Aliabadi, S., Behr, M., Johnson, A., Kalro, V. & Litke, M. Flow simulation and high performance computing. *Computat. Mech.* **18**, 397–412 (1996), doi: 10.1007/bf00350249.
38. Sanbonmatsu, K. Y. & Tung, C.-S. High performance computing in biology: multimillion atom simulations of nanoscale systems. *J. Struct. Biol.* **157**, 470–480 (2007), doi: 10.1016/j.jsb.2006.10.023.
39. Allcock, B. *et al.* Data management and transfer in high-performance computational grid environments. *Parallel Comput.* **28**, 749–771 (2002), doi: 10.1016/s0167-8191(02)00094-7.
40. Mavridis, I. & Karatza, H. Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark. *J. Syst. Softw.* **125**, 133–151 (2017), doi: 10.1016/j.jss.2016.11.037.

41. El-Haija, S. A., Kothari, N., Lee, J., Natsev, P., Toderici, G., Varadarajan, B. & Vijayanarasimhan, S. YouTube-8M: a large-scale video classification benchmark. (2016), https://arxiv.org/abs/1609.08675.
42. Lee, C. A., Gasster, S. D., Plaza, A., Chang, C.-I. & Huang, B. Recent developments in high performance computing for remote sensing: a review. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **4**, 508–527 (2011), doi: 10.1109/jstars.2011.2162643.
43. O'Driscoll, A., Daugelaite, J. & Sleator, R. D. 'Big data', Hadoop and cloud computing in genomics. *J. Biomed. Inform.* **46**, 774–781 (2013), doi: 10.1016/j.jbi.2013.07.001.
44. Dean, J. & Ghemawat, S. MapReduce. *Commun. ACM* **51**, 107 (2008), doi: 10.1145/1327452.1327492.
45. Stegmaier, J. New methods to improve large-scale microscopy image analysis with prior knowledge and uncertainty. (2017), doi: 10.5445/ksp/1000060221.
46. Goscinski, W. J. *et al.* The multi-modal Australian ScienceS Imaging and Visualization Environment (MASSIVE) high performance computing infrastructure: applications in neuroscience and neuroinformatics research. *Front. Neuroinform.* **8**, (2014), doi: 10.3389/fninf.2014.00030.
47. Knuth, D. E. Structured programming with go to statements. *ACM Comput. Surv.* **6**, 261–301 (1974), doi: 10.1145/356635.356640.
48. Walt, S., Colbert, S. C. & Varoquaux, G. The NumPy array: a structure for efficient numerical computation. *Comput. Sci. Eng.* **13**, 22–30 (2011), doi: 10.1109/mcse.2011.37.
49. Larsen, S. & Amarasinghe, S. Exploiting superword level parallelism with multimedia instruction sets. *ACM Sigplan Notices* **35**, 145–156 (2000), doi: 10.1145/358438.349320.
50. Thadani, M. & Khalidi, Y. Y. An Efficient Zero-Copy I/O Framework for UNIX®. (1995).
51. Stancevic, D. Zero Copy I: user-mode perspective. (2003). Retrieved from https://www.linuxjournal.com/article/6345?page=0,0.
52. Kelly, P. H. Advanced computer architecture: caches and memory systems. (2003). Retrieved from https://pdf.semanticscholar.org/12db/961cda6f5adaccb5731fcce0a0044752f3d1.pdf.
53. BLAS (Basic Linear Algebra Subprograms). (2017). Retrieved from http://www.netlib.org/blas/.
54. PyTorch. (2018). Retrieved from https://pytorch.org/.
55. Intel. Intel® Math Kernel Library. (2018). Retrieved from https://software.intel.com/en-us/mkl.
56. Numba: A High-Performance Python Compiler. (2018). Retrieved from http://numba.pydata.org/.
57. Godbolt, M. Compiler Explorer. Retrieved from https://godbolt.org/.
58. Data Analysis Unit, European Synchrotron Radiation Facility, Grenoble. General introduction to PyFAI. (2018). Retrieved from https://pyfai.readthedocs.io/en/latest/pyFAI.html.
59. Alted, F. Why modern CPUs are starving and what can be done about it. *Comput. Sci. Eng.* **12**, 68–71 (2010), doi: 10.1109/mcse.2010.51.
60. Intel High Performance Data Division. Architecting a High Performance Storage System. (2014). Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/architecting-lustre-storage-white-paper.pdf.

61. Patel, A. B., Birla, M. & Nair, U. Addressing big data problem using Hadoop and Map Reduce. In *2012 Nirma University International Conference on Engineering (NUiCONE)* (IEEE, 2012), doi: 10.1109/nuicone.2012.6493198.
62. Shvachko, K., Kuang, H., Radia, S. & Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th Symp. Mass Storage Systems and Technologies (MSST)* (IEEE, 2010), doi: 10.1109/msst.2010.5496972.
63. Lustre Software Release 2.x Operations Manual. (2018).
64. Apache Software Foundation. HDFS short-circuit local reads. (2018). Retrieved from https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.
65. Cooley, J. W. & Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **19**, 297–301 (1965), doi: 10.1090/s0025-5718-1965-0178586-1.
66. Zaharia, M. *et al.* Apache spark. *Commun. ACM* **59**, 56–65 (2016), doi: 10.1145/2934664.
67. Rocklin, M. Dask: parallel computation with blocked algorithms and task scheduling. In *Proc. 14th Python in Science Conf.* SciPy. (2015), doi: 10.25080/majora-7b98e3ed-013.
68. Foust, G., Järvi, J. & Parent, S. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proc. 2015 ACM SIGPLAN Int. Conf. Generative Programming: Concepts and Experiences - GPCE 2015* (ACM Press, 2015), doi: 10.1145/2814204.2814207.
69. Abramov, D. Redux: core concepts. (2018). Retrieved from https://redux.js.org/introduction/core-concepts.
70. Abramov, D. The case for flux. (2015). Retrieved from https://medium.com/swlh/the-case-for-flux-379b7d1982c6.
71. Taylor, R. N. *et al.* A component- and message-based architectural style for GUI software. *IEEE Trans. Softw. Eng.* **22**, 390–406 (1996), doi: 10.1109/32.508313.
72. Rogic, I. React, redux and immutable.js: ingredients for efficient web applications. Retrieved from https://www.toptal.com/react/react-redux-and-immutablejs.
73. Shneiderman, B., Plaisant, C., Cohen, M. & Jacobs, S. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, 5th Edition* (Pearson, 2009). Retrieved from https://onlinelibrary.wiley.com/doi/full/10.1002/asi.21215.
74. Naul, B., Walt, S., Crellin-Quick, A., Bloom, J. S. & Pérez, F. Cesium: open-source platform for time-series inference. (2016).
75. Facebook Inc. React: a JavaScript library for building user interfaces. (2018). Retrieved from https://reactjs.org/.
76. Abramov, D. Redux: a predictable state container for JavaScript apps. (2018). Retrieved from https://redux.js.org/.
77. Abramov, D. Redux: usage with react. (2018). Retrieved from https://redux.js.org/basics/usage-with-react.
78. Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* (2014). Retrieved from https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.
79. Bernstein, D. Containers and cloud: from LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **1**, 81–84 (2014), doi: 10.1109/mcc.2014.51.
80. Schindelin, J. *et al.* Fiji: an open-source platform for biological-image analysis. *Nature Methods* **9**, 676–682 (2012), doi: 10.1038/nmeth.2019.

81. Shaw, M. Prospects for an engineering discipline of software. *IEEE Softw.* **7**, 15–24 (1990), doi: 10.1109/52.60586.
82. Zuiderwijk, A., Jeffery, K. & Janssen, M. The potential of metadata for linked open data and its value for users and publishers. *JeDEM* **4**, 222–244 (2012), doi: 10.29379/jedem.v4i2.138.
83. The HDF Group. High level introduction to HDF5. (2016). Retrieved from https://support.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf.
84. The HDF Group. Enterprise support. (2018). Retrieved from https://www.hdfgroup.org/solutions/enterprise-support/.
85. Willinsky, J. The unacknowledged convergence of open source, open access, and open science. *First Monday* **10**, (2005), doi: 10.5210/fm.v10i8.1265.
86. Okoli, C. & Nguyen, J. Business models for free and open-source software. *SSRN Electron. J.* (2015), doi: 10.2139/ssrn.2568185.
87. NumFOCUS. Better tools to build a better world. (2018). Retrieved from https://numfocus.org/.
88. The APACHE Software Foundation. (2018). Retrieved from https://www.apache.org/.
89. The Linux Foundation. (2018). Retrieved from https://www.linuxfoundation.org/.
90. Microsoft. Microsoft Open Source. (2018). Retrieved from https://opensource.microsoft.com/.
91. Google. Google Open Source. (2018). Retrieved from https://opensource.google.com/.
92. Facebook. Facebook Open Source: building community through open source technology. (2018). Retrieved from https://opensource.fb.com/.
93. Clausen, A. *et al.* Libertem/Libertem: 0.1.0. Zenodo. (2018), doi: 10.5281/zenodo.1477847.
94. Williams, D. B. & Carter, C. B. The transmission electron microscope. In *Transmission Electron Microscopy* (Springer US, 1996), pp. 3–17, doi: 10.1007/978-1-4757-2519-3_1.
95. Goto, K. & Geijn, R. A. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* **34**, 1–25 (2008), doi: 10.1145/1356052.1356053.
96. Mehta, P. *et al.* Comparative evaluation of big-data systems on scientific image analytics workloads. (2016).
97. Anaconda Inc. Comparison to spark. (2018). Retrieved from http://docs.dask.org/en/latest/spark.html.
98. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A. & Pouchet, L.-N. Polly-polyhedral optimization in LLVM. Retrieved from http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-07.pdf.
99. Anaconda, Inc. Journey of a task. (2018). Retrieved from https://distributed.dask.org/en/latest/journey.html.
100. The LLVM Foundation. The LLVM Compiler Infrastructure. (2019). Retrieved from The LLVM Compiler Infrastructure: https://llvm.org/.
101. Lemire, D. & Boytsov, L. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* **45**, 2015 (2012), doi: 10.1002/spe.2203.
102. Könnecke, M. *et al.* The NeXus data format. *J. Appl. Crystallog.* **48**, 301–305 (2005), doi: 10.1107/s1600576714027575.
103. Axboe, J. Flexible I/O Tester. Retrieved from https://github.com/axboe/fio.
104. Axboe, J. Add io_uring IO interface. (2018). Retrieved from https://patchwork.kernel.org/patch/10806677/.

105. Caron, J. *Model-Based Reconstruction of Magnetisation Distributions in Nanostructures from Electron Optical Phase Images* E-Book, Vol. 177 (Forschungszentrum, Zentralbibliothek, Jülich, 2018). Retrieved from http://juser.fz-juelich.de/record/851773.
106. Song, D. *et al.* Quantification of magnetic surface and edge states in an FeGe nanostripe by off-axis electron holography. *Phys. Rev. Lett.* **120**, (2018), doi: 10.1103/physrevlett.120.167204.