Title

Accelerating x-ray tracing for exascale systems using Kokkos

Permalink

https://escholarship.org/uc/item/38v3k44n

Journal

Concurrency and Computation Practice and Experience, 36(5)

ISSN

1532-0626

Authors

Wittwer, Felix
Sauter, Nicholas K
Mendez, Derek
et al.

Publication Date

2024-02-29

DOI

10.1002/cpe.7944

Copyright Information

Accelerating X-Ray Tracing for Exascale Systems using Kokkos

Felix Wittwer*[1] | Nicholas K. Sauter[2] | Derek Mendez[2] | Billy K. Poon[2] | Aaron S. Brewster[2] | James M. Holton[2] | Michael E. Wall[3] | William E. Hart[4] | Deborah J. Bard[1] | Johannes P. Blaschke[1]

[1] National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2] Molecular Biophysics and Integrated Bioimaging Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[3] Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, NM, USA
[4] Sandia National Laboratories, Albuquerque, NM, USA

Correspondence
*Corresponding author, Email: fwittwer@lbl.gov

Summary

The upcoming exascale computing systems Frontier and Aurora will draw much of their computing power from GPU accelerators. The hardware for these systems will be provided by AMD and Intel, respectively, each supporting their own GPU programming model. The challenge for applications that harness one of these exascale systems will be to avoid lock-in and to preserve performance portability.

We report here on our results of using Kokkos to accelerate a real-world application on NERSC's Perlmutter Phase 1 (using NVIDIA A100 accelerators) and Crusher, the testbed system for OLCF's Frontier (using AMD MI250X). By porting to Kokkos, we successfully ran the same X-ray tracing code on both systems and achieved speed-ups between 13 % and 66 % compared to the original CUDA code. These results are a highly encouraging demonstration of using Kokkos to accelerate production science code.

KEYWORDS:
cross compilation, code optimization, Kokkos, Nvidia GPU, AMD GPU

1 | INTRODUCTION

The upcoming high-performance computing (HPC) systems Frontier and Aurora will be the first exascale machines. Both are capable of performing more than 10^{18} floating point operations per second. The majority of this computing power comes from the GPU accelerators of these systems. Due to their massive parallelism, GPUs are well suited for repetitive tasks.

Using GPUs requires vendor specific programming models, such as CUDA for NVIDIA or HIP for AMD. This makes portability between different systems challenging. As an alternative, programming models such as OpenMP offloading or Kokkos[1] provide an abstraction layer between the source code and the GPU hardware. With these abstraction layers, the same code can be compiled for different architectures, thus combining portability with high performance.

2 | SCIENTIFIC BACKGROUND

X-ray crystallography is an indispensable tool to study the structure of molecules, with applications ranging from materials science[2] to understanding the function of proteins[3]. In crystallography, small crystals of an unknown sample (e.g. a protein whose molecular structure is to be determined) are placed in an x-ray beam. By measuring how the x-rays are scattered, the position of each atom in the sample can be determined. A major application of x-ray crystallography is determining molecular structures of proteins. Knowing the protein structure is crucial to understand how a protein interacts. From this understanding, drugs can be designed to, e.g., treat infections by blocking specific interactions.

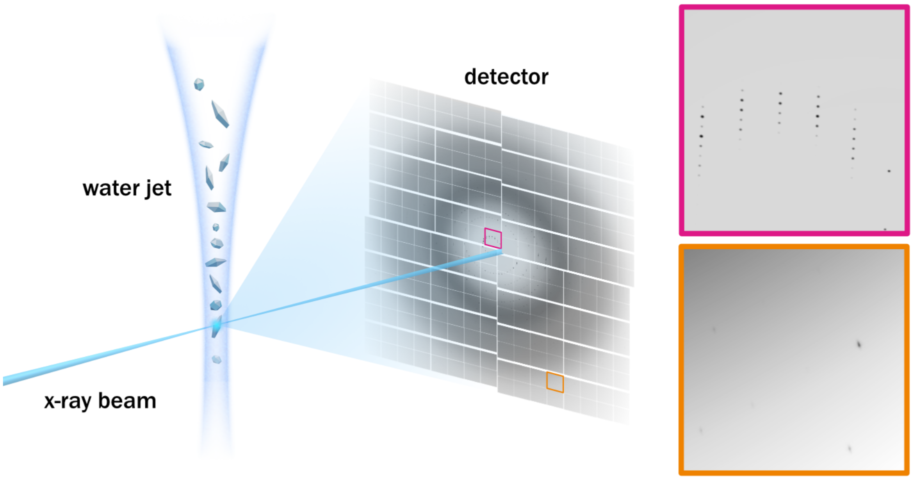

Figure 1 *Left:* Experimental setup for serial femtosecond x-ray crystallography (SFX). A water jet delivers the protein crystals to the x-ray beam. For each x-ray pulse, the imaging detector records the scattered x-rays. *Right:* Two detector close-ups. Top: Small scattering angles close to the beam axis. Bottom: Large scattering angles at the edge of the detector.

Because the molecules inside the crystals are arranged in a long-range repeating order, the crystals scatter x-rays only into certain directions – i.e., scattering only occurs at those angles, which allow for positive interference from the x-rays scattered by different atoms. The x-rays scattered into these discrete directions form Bragg spots on the imaging detector (the right panels in Fig. 1 illustrate one possible pattern). Mathematically speaking, a molecular structure is represented using a list of complex numbers called structure factors, where each Bragg spot is associated with one such structure factor. Reconstructing the protein structure is therefore a two step process: first, the structure factor amplitudes are determined from the Bragg spots. Then, they are used to reconstruct the protein structure by Fourier transformation. If more Bragg spot measurements are available for the reconstruction, the atomic positions can be determined more accurately. Generally, larger crystals or a more intense x-ray beam are required to increase the signal of weak Bragg spots. However, many protein crystals are challenging to crystallize and grow only to microscopic size. Scientists therefore opt for increasing beam intensities. However, as proteins are sensitive to radiation damage, they are quickly destroyed by high intensity x-ray beams.

Serial femtosecond x-ray crystallography (SFX) avoids the problem of radiation damage by using ultra-short x-ray pulses from x-ray free-electron lasers (XFELs), called "shots". In essence, the pulses are so short that the Bragg spots can be measured before beam damage has had time to degrade the sample. Each x-ray pulse lasts only a few tens of femtoseconds, fast enough to freeze all atomic motion and capture a snapshot of the crystal before any damage becomes visible. Still, this process will destroy the sample, so fresh crystals must be constantly fed in before each x-ray pulse, see Fig. 1. Measuring a complete dataset requires tens of thousands of crystals, each producing only a single scattering image.

Most crystallography methods determine the structure factor amplitudes by simply integrating the number of photons in each Bragg spot. Apart from the structure factors, the intensity of a Bragg spot is also influenced by the orientation of the crystal, how the crystal is composed from smaller mosaic blocks, and how the energy spectrum of the photons fluctuates from shot to shot. By averaging multiple images, each under different conditions, these influences can be averaged out. For SFX, this requires tens of thousands of scattering images.

Instead of simply integrating each Bragg spot, x-ray tracing aims to simulate the photon intensity of every detector pixel by creating a physical model of each crystal[4]. Modeling each crystal is an iterative process, in which the current parameter estimates are refined until the simulated scattering image matches the measured one. By recovering the unknown parameters from each crystal, x-ray tracing can accurately determine the structure factors from an order of magnitude fewer scattering images. As measurement time at XFELs is scarce, this allows scientists to study more samples or more experimental conditions during the same experiment.

However, while x-ray tracing may require fewer experimental resources, the computational effort to accurately model each pixel grows considerably in comparison to conventional methods. At the same time, quick feedback on the quality of the collected data is critical for XFEL experiments[5]. X-ray tracing therefore faces the challenge of performing a more complex data analysis in less time. Providing XFEL experiments with quick-turnaround analysis of the terabytes of data requires HPC at the Exascale.

3 | GPU ACCELERATION WITH KOKKOS

To simulate scattering images from a physical model, we have developed the program *nanoBragg*[6], which is part of the CCTBX software suite[7,8]. CCTBX consists of a high-level workflow written in Python and accelerated kernels written in C++ (interfacing with Python via *Boost.Python*). This approach has been proven to be highly successful in analyzing large data sets using HPC resources[5,9]. By designing *nanoBragg* as a kernel compatible with the CCTBX workflow, we see it as the first building block of a larger pixel-level XFEL data analysis workflow.

In x-ray tracing, each detector pixel can be simulated nearly independently of all other pixels, making x-ray tracing ideally suited to exploit the massive parallelism of modern graphic processing units (GPUs). Compared to a 64-core AMD Milan CPU, an NVIDIA A100 GPU is able to simulate diffraction images ten times faster, using the Kokkos implementation of *nanoBragg*, and comparing the OpenMP and CUDA backends for Kokkos, respectively. *nanoBragg* uses MPI (specifically *mpi4py*) to distribute tasks (batches of images to simulate) to multiple nodes and GPUs using MPI. Each GPU then performs work independently. Solving for an unknown molecular structure therefore follows a fork-join parallelism, where each GPU simulates a batch of parameters independently, followed by a MPI reduction[4]. The *nanoBragg* benchmark discussed here forgoes the final reduction step, instead saving the simulated images to the file system.

The upcoming exascale systems Frontier (OLCF) and Aurora (ALCF) are essential to obtain rapid feedback during XFEL experiments. However, the GPUs for these systems will be provided by AMD, and Intel, respectively. As both vendors supply their own alternative to CUDA, *nanoBragg* must be adapted to each system. To avoid code duplication and site-specific optimizations, we decided to use Kokkos[1].

3.1 | Kokkos

Kokkos is a C++ programming model that allows a single code base to target different HPC platforms[1]. To achieve this, Kokkos implements abstract memory and execution spaces. Only during the compilation are the abstract spaces and commands converted to CUDA, HIP, OpenMP, etc. This way, the same Kokkos code can be used on systems with different GPUs or even systems with no GPU.

Each execution space is associated with a particular memory space. While the execution space defines where the computation is done, the computation itself is defined by *execution patterns*, which are the Kokkos equivalent to kernels in CUDA. Kokkos implements three patterns:

- `parallel_for`: Corresponds to a for-loop where each iteration is executed independently, for example element-wise addition of two vectors.
- `parallel_reduce`: A reduction that collects and combines the result of all iterations, for example finding the longest word in a list.
- `parallel_scan`: A combination of the former two that uses multiple reductions, for example calculating a histogram of an image.

The typical use case for execution patterns is to work on large data arrays. Kokkos provides its own data structure called *View*. A *View* is a multi-dimensional array which can be transferred between different memory spaces and layouts. The last one is important, e.g. in matrix multiplications, the best performance on a multi-core CPU might be achieved with a row-major layout and on a GPU with a column-major layout. Kokkos automatically manages the conversion between layouts when a *View* is transferred between different memory spaces. This makes it easy to first initialize a *View* on the CPU and then transfer it to GPU memory.

3.2 | Porting nanoBragg to Kokkos

The task of porting *nanoBragg* can be split into two parts: The first part is converting the CUDA kernels to Kokkos execution patterns, the second part is replacing all CUDA arrays with Kokkos *Views*. As there is no interaction between the individual detector pixels, all kernels can be replaced with the parallel_for pattern. Most of the computational work in *nanoBragg* is done in three kernels:

- *nanoBraggSpots*: The most complex kernel, containing about 350 lines of code. This kernel simulates the Bragg spots. It takes into account crystal orientation, mosaicity and photon energy.
- *addBackground*: The second most complex kernel with about 120 lines of code. This kernel simulates the background scattering from the air and water that surround the crystal.
- *addArray*: A simple kernel that adds the results from the other two kernels.

We illustrate the porting process by using a simplified version of the *addArray* kernel:

```
__global__
void addArray(double* lhs, float* rhs, int size) {
  int j = blockDim.x * blockIdx.x + threadIdx.x;
  if (j < size) {
    lhs[j] = lhs[j] + (double) rhs[j];
  }
}
```

Listing 1: The *addArray* kernel in CUDA

For the Kokkos version, we replaced the *lhs* and *rhs* arrays with *Views* and the body of the kernel with a parallel_for pattern. As the memory management of *Views* is done by Kokkos, we could remove the custom-written CUDA memory management. The parallel_for execution pattern takes three arguments: an individual name for debugging and profiling purposes, an execution policy and a functor that holds the computational work. In simple cases like this, we make use of the basic execution policy that simply iterates over *size* number of elements. The functor can be integrated into the pattern by using lambda expressions, this way the structure mirrors how kernels are defined:

```
void addArray(Kokkos::View<double*> lhs, Kokkos::View<float*> rhs, int size) {
  Kokkos::parallel_for("addArray", size, KOKKOS_LAMBDA (const int& j) {
    lhs(j) = lhs(j) + (double) rhs(j);
  });
}
```

Listing 2: The *addArray* kernel in Kokkos

The other kernels were converted much in the same way, with a few challenges. One of these is an edge case when using lambda expressions for device code in member functions of a class. Suppose *init()* is a member function that initializes the view `data` with a certain value:

```
void Container::init() {
  Kokkos::parallel_for("init", size, KOKKOS_LAMBDA (const int& j) {
    data(j) = m_value;
  });
}
```

Listing 3: Lambda capture

Since they are class members, the compiler replaces `data` and `m_value` during the compilation with `this->data` and `this->m_value`. The problem arises from `this` being a pointer in CPU memory. If CUDA is used, this creates an illegal memory access when the GPU tries to access the pointer.

There are multiple ways around this issue: One way is to avoid lambda expressions in this situation and instead explicitly define the functors. Another option is to copy the specific member variables into local variables before the execution pattern is called. And finally, in C++17 lambda expressions have been extended so they can explicitly capture the object, not the pointer. Unfortunately, the CCTBX code is currently not compatible with C++17. Therefore, we choose to use local variables in these situations and otherwise minimize the use of kernels in member functions.

4 | PERFORMANCE AND PORTABILITY

The goal of porting the *nanoBragg* code to Kokkos is to achieve portability between the upcoming Frontier and Aurora systems as well as Perlmutter. We test the portability on Perlmutter Phase 1 and Crusher, the latest Frontier testbed system. These systems are not production resources, the performance numbers given here were not always reproducible and will be different from the final systems.

Increasing the portability of a code can potentially reduce the performance. We therefore also assess on Perlmutter the performance of the Kokkos port in comparison to the original CUDA version. On Perlmutter, we use the CUDA backend of Kokkos and on Crusher accordingly the HIP backend.

Each node of Perlmutter Phase 1 is equipped with an AMD EPYC 7763 CPU and four NVIDIA A100 GPUs. On Crusher each node is equipped with an AMD EPYC 7A53 CPU and four AMD MI250X GPUs. Each MI250X GPU is equipped with two graphic compute dies (GCDs) totalling eight GCDs per node. These eight GCDs function effectively like eight separate GPUs per node.

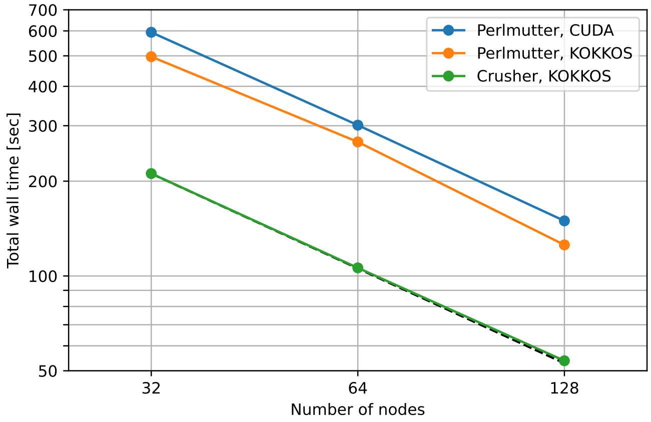

Figure 2 Strong scaling of *nanoBragg* for the simulation of 100 000 scattering images. The dashed line shows ideal scaling where doubling the number of nodes halves the simulation time. On Perlmutter Phase 1, 4 MPI ranks per node are used and Kokkos is used with the CUDA backend. On Crusher, 8 MPI ranks per node are used and Kokkos is used with the HIP backend.

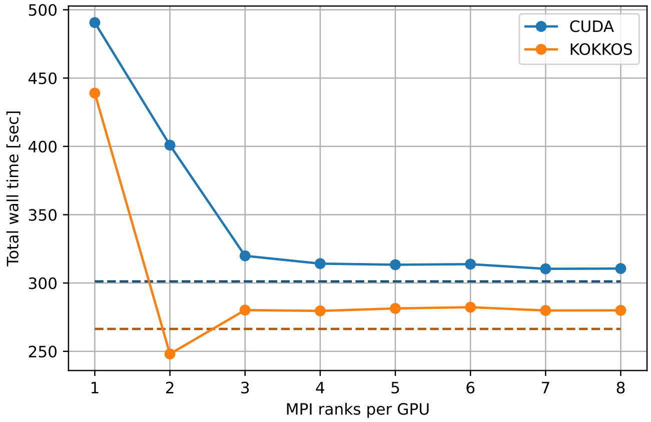

Figure 3 Time to simulate 100 thousand diffraction patterns using 64 Perlmutter nodes. In contrast to Fig. 2, this benchmark includes the time to save each image to the file system. For comparison, the previous times without file saving are indicated with dashed lines. As the filesystem performance on Perlmutter can vary (due to it not being a production system) at the time of writing, this graph shows the best performance over several repeated runs.

As a benchmark, we tested a scenario that simulates 100 000 scattering patterns. Due to performance fluctuations of the file system (a consequence of Perlmutter Phase 1 and Crusher not being production systems), we did not include the saving of the simulated images into the benchmark. The tests were performed using 32, 64, and 128 nodes, all results are plotted in Fig. 2. On Perlmutter Phase 1, we used 4 MPI ranks per node, one for each NVIDIA A100 GPU. On Crusher, we used 8 MPI ranks per node, one for each MI250X GCD. Every test case shows almost ideal scaling behavior, highlighting how x-ray tracing benefits from parallel computing. On Perlmutter, the Kokkos version is about 13 % faster than the original CUDA version, indicating no loss in performance due to the port to Kokkos. Concerning portability, Kokkos allows *nanoBragg* to exploit the hardware of Crusher for a more than 60 % faster performance per node, compared to Perlmutter Phase 1.

Next we re-enable file I/O. This introduces workflow latency and dependence on a shared resource (the file system). We observe that the strong scaling shown in Fig. 2 is preserved – albeit shifted up due to the increased time spent on I/O. CCTBX seeks to hide workflow latency by assigning more than one MPI rank to each GPU. We observe that this is a largely successful strategy (as total runtime decreases with increasing number of MPI ranks per GPU). Fig. 3 shows the total runtime to trace 100 000 images using 64 Perlmutter nodes. We see that increasing the number of ranks per GPU increases the total throughput. The throughput reaches a plateau when 3 or more ranks per GPU are used. Interestingly, the CUDA implementation achieves the highest throughput when many (approx. 7 or greater) ranks are sharing the same GPU. In contrast, the Kokkos implementation achieves the maximum throughput when only two MPI ranks share the same GPU. For this configuration, the kernel runtimes coincide, allowing the GPU to seamlessly alternate between the two ranks with no overlap or dead time.

Furthermore, comparing Fig. 2 and Fig. 3, we see that the increased latency due to I/O can be hidden by sharing GPUs accross ranks.

5 | KERNEL PROFILING

To determine why the Kokkos version performs better on Perlmutter than the original CUDA version, we used Nsight Systems and Nsight Compute to profile both versions. For the three main kernels, the execution times on a NVIDIA A100 GPU are given in Table 1. For all three kernels, Kokkos

Table 1 Kernel run-times.

	nanoBraggSpots	addBackground	addArray
CUDA	8.28 ms	1.87 ms	0.13 ms
Kokkos	6.98 ms	1.76 ms	0.12 ms
Speed-up	+15.7 %	+5.9 %	+7.7 %

Table 2 nanoBraggSpots details.

	CUDA	Kokkos
Run-time	8.28 ms	6.98 ms
Compute Throughput	65.05 %	77.42 %
Memory Throughput	21.05 %	21.35 %
Registers	130	116
Theoretical Occupancy	18.75 %	25 %
Achieved Occupancy	16.8 %	24.74 %

achieves a speed-up of more than five percent. As the *nanoBraggSpots* kernel accounts for most of the GPU computation time, the speed-up of more than 15 % for this kernel has a significant impact on the overall run time. On the other end of the complexity spectrum is the *addArray* kernel. While it is one of the smallest kernels and any improvements will only have a small influence in the grand scheme, achieving a speed-up of nearly eight percent even for short kernels is remarkable.

Using Nsight Compute we studied the critical *nanoBraggSpots* kernel in detail. The results are summarized in Table 2. In the table, the compute and memory throughput give an overview how much of the compute and memory resources of the GPU are utilized by the *nanoBraggSpots* kernel. The performance difference between CUDA and Kokkos is mostly a result of the higher compute throughput of the Kokkos kernel, the memory throughput is nearly identical. The increased compute throughput in Kokkos is made possible by using fewer registers. Each multiprocessor on an A100 GPU has a total of 65 536 32 bit-registers for a maximum of 2048 simultaneous threads (64 warps per multiprocessor times 32 threads per warp). However, the GPU can only run at full occupancy if each thread uses less then 65535/2048=32 registers. With 130 registers, the A100 GPU can run at most 12 warps out of the 64 available. The Kokkos kernel uses 116 registers, which is below the critical threshold of 128 registers, and can therefore run a maximum of 16 warps, 33 % more.

The situation for the other kernels is similar. The *addBackground* kernel goes from 96 to 80 registers and the *addArray* kernel goes from 25 to 16 registers. Kokkos uses consistently fewer registers than the original CUDA implementation, allowing a higher occupancy and throughput.

6 | CONCLUSION

In this paper, we have reported on our work to port *nanoBragg* to Kokkos. Starting from a CUDA code base, most of the port was straightforward. Arising difficulties with the port were solved. We have demonstrated that the performance of the code did not suffer from this port and has even increased by 13 %. We have also successfully demonstrated performance portability between Perlmutter Phase 1 (NERSC) and Crusher (OLCF). We are currently in the process of testing the Aurora test system (ALCF). Until now, we have not used advanced Kokkos features such as nested parallelism, which should further increase the performance.

As *nanoBragg* is representative of many scientific codes, our reported findings indicate that Kokkos is a powerful tool to not only achieve performance portability in real-world applications, but also to accelerate existing codes. Moreover, we are able to integrate Kokkos into existing scientific workflows without needing to port an entire code base. This allows scientists to make progress, even if staff time is limited.

ACKNOWLEDGMENTS

N.K.S, J.P.B., M.E.W., F.W. and D.B. acknowledge support from the Exascale Computing Project (grant 17-SC-20-SC), a collaborative effort of the Department of Energy (DOE) Office of Science and the National Nuclear Security Administration. This research used resources of the National

Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Conflict of interest

The authors declare no potential conflict of interests.[?]

References

1. Trott CR, Lebrun-Grandie D, Arndt D, et al. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Trans. Parallel Distrib. Syst.* 2022; 33(4): 805–817. doi: 10.1109/TPDS.2021.3097283

2. Schriber EA, Paley DW, Bolotovsky R, et al. Chemical crystallography by serial femtosecond X-ray diffraction. *Nature* 2022; 601(7893): 360–365. doi: 10.1038/s41586-021-04218-3

3. Keable SM, Kölsch A, Simon PS, et al. Room temperature XFEL crystallography reveals asymmetry in the vicinity of the two phylloquinones in photosystem I. *Scientific Reports* 2021; 11(1): 21787. doi: 10.1038/s41598-021-00236-3

4. Mendez D, Bolotovsky R, Bhowmick A, et al. Beyond Integration: Modeling Every Pixel to Obtain Better Structure Factors from Stills. *IUCrJ* 2020; 7(6): 1151–1167. doi: 10.1107/S2052252520013007

5. Blaschke JP, Brewster AS, Paley DW, et al. Real-Time XFEL Data Analysis at SLAC and NERSC: A Trial Run of Nascent Exascale Experimental Data Analysis. *Concurrency and Computation: Practice and Experience.* accepted.

6. Sauter NK, Kern J, Yano J, Holton JM. Towards the Spatial Resolution of Metalloprotein Charge States by Detailed Modeling of XFEL Crystallographic Diffraction. *Acta Crystallogr. D* 2020; 76(2): 176–192. doi: 10.1107/S2059798320000418

7. Grosse-Kunstleve RW, Sauter NK, Moriarty NW, Adams PD. The Computational Crystallography Toolbox: Crystallographic algorithms in a reusable software framework. *Journal of Applied Crystallography* 2002; 35(1): 126–136. doi: 10.1107/S0021889801017824

8. Sauter NK, Hattne J, Grosse-Kunstleve RW, Echols N. New Python-based methods for data processing. *Acta Crystallographica Section D* 2013; 69(7): 1274–1282. doi: 10.1107/S0907444913000863

9. Giannakou A, Blaschke JP, Bard D, Ramakrishnan L. Experiences with Cross-Facility Real-Time Light Source Data Analysis Workflows. In: *2021 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*; 2021; St. Louis, MO, USA: 45–53