

# Optimizing and Scaling the 3D Reconstruction of Single-Particle Imaging

Niteya Shah  
Dept. of Computer Science  
Virginia Tech  
Blacksburg, VA, USA  
niteya@vt.edu

Christine Sweeney  
Applied Computer Science Group  
Los Alamos National Laboratory  
Los Alamos, NM 87545 USA  
cahrens@lanl.gov

Vinay Ramakrishnaiah  
Applied Computer Science Group  
Los Alamos National Laboratory  
Los Alamos, NM 87545 USA  
vinayr@lanl.gov

Jeffrey Donatelli  
Center for Advanced Mathematics for Energy Research Applications  
Lawrence Berkeley National Laboratory  
Berkeley, CA, USA  
jjdonatelli@lbl.gov

Wu-chun Feng  
Dept. of Computer Science  
Virginia Tech  
Blacksburg, VA, USA  
wfeng@vt.edu

**Abstract**—An X-ray free electron laser (XFEL) facility can produce on the order of 1,000,000 extremely bright X-ray light pulses per second. Using an XFEL to image the atomic structure of a molecule requires fast analysis of an enormous amount of data, estimated to exceed one terabyte per second and requiring petabytes of storage. The SpiniFEL application provides such analysis by determining the 3D structure of proteins from single-particle imaging (SPI) experiments performed using XFELs, but it needs significantly better performance and efficiency to scale and keep up with the terabyte-per-second data production. Thus, this paper addresses the high-performance computing optimizations and scaling needed to improve this 3D reconstruction of SPI data. First, we optimize data movement, memory efficiency, and algorithms to improve the *per-node* computational efficiency and deliver a 5.28× speedup over the baseline GPU implementation.

In addition, we achieved a 485× speedup for the post-analysis reconstruction resolution, which previously took as long as the 3D reconstruction of SPI data. Second, we present a novel distributed shared-memory computational algorithm to hide data latency and load-balance network traffic, thus enabling the processing of 128× more orientations than previously possible. Third, we conduct an exploratory study over the hyperparameter space for the SpiniFEL application to identify the optimal parameters for our underlying target hardware, which ultimately led to an up to 1.25× speedup for the number of streams. Overall, we achieve a 6.6× speedup (i.e.,  $5.28 \times 1.25$ ) over the previous fastest GPU-MPI-based SpiniFEL realization.

## I. INTRODUCTION

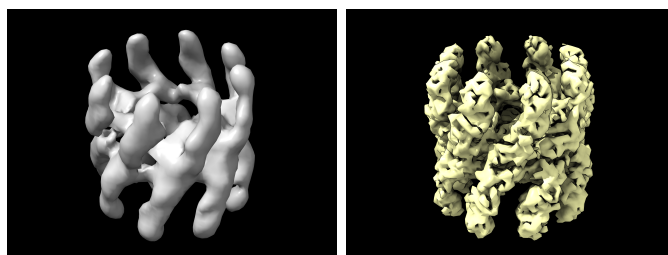
The ExaFEL project from the Exascale Computing Project (ECP) [1] facilitates the reconstruction of macromolecules using the Linac Coherent Light Source (LCLS) [2] and the upgraded LCLS-II [3]. LCLS and LCLS-II are X-ray free electron laser (XFEL) facilities that enable the study of the structure and dynamics of single macromolecules and increase the repetition rate of the X-ray source from 120 pulses per

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and in part by the Synergistic Environments for Experimental Computing (SEEC) Center at Virginia Tech.

second to 1,000,000 pulses per second, respectively. With the 1,000,000 pulses per second generating a terabyte of data per second (1 TB/s), scientists need fast analysis of the data to keep up with this increased data rate and, in turn, create 3D reconstruction of single-particle imaging in near real-time.

While ExaFEL’s data analysis workflow for serial femtosecond crystallography assists researchers in observing the dynamic movement of atoms, the SpiniFEL project aims to “break free from the need for crystallization and allow for imaging molecular dynamics at near-ambient conditions” [4]. The SpiniFEL code [5] reconstructs the electron density map of a biological macromolecule from single-particle imaging (SPI) [6] diffraction data. This diffraction data contains diffraction patterns collected by a detector that captures diffracted light after the beam hits the macromolecule. Sample molecules are streamed in front of the X-ray beam, and after diffracting the light, the molecule is destroyed by it. Each molecule has a different 3D orientation when it is hit by the beam.

Fig. 1 shows the 3D structure of the 3IYF protein, reconstructed using SpiniFEL, resolved at 13.7 Å, and visualized with ChimeraX [7]. SpiniFEL’s analysis process is both computationally intensive and data-intensive. Diffraction patterns need to be sorted into conformations and oriented in 3D for model building and refinement. Hundreds of thousands of



(a) Aligned structure.

(b) Reference structure.

Fig. 1: Reconstruction of the 3IYF protein.

diffraction patterns may need to be analyzed to produce a high-resolution result. Additionally, SpiniFEL aims to run in near real-time *while* an experiment is taking place so that the feedback about the data can guide the data collection strategy. This, in turn, enables experimentalists to study molecules that are difficult to crystallize, thus creating further opportunities for discovery.

The work presented here delivers both GPU- and MPI-based optimizations for SPI that were developed by profiling the application to identify key bottlenecks that appear as we increase the complexity of the problem. In addition, we overcome a memory capacity limitation in one of the components of the workflow that arises when replicating the model diffraction images  $I(\mathbf{q}_k)$  at their various orientations for all the ranks. Finally, we study the behavior of the application with its many hyperparameters to understand its behavior over different problem complexities. In all, we deliver the following research contributions to this application:

- GPU-optimized algorithms that improve the computational efficiency and throughput of the application, leading to an overall  $5.28\times$  speedup over the baseline GPU implementation.
- A novel distributed-memory data structure and a corresponding low contention MPI-RMA access algorithm for performing pipelined symmetric all-to-all communication between ranks of multi-GPU nodes, leading to a  $1.25\times$  speedup when using multiple streams for the optimized GPU implementation.
- Exploration of avenues for optimizing performance on upcoming architectures having higher memory capacity and bandwidth through adjustments to the batch size hyperparameter.

## II. BACKGROUND

In this section, we provide background about single-particle imaging (SPI) and its associated workflow as well as hardware and software considerations, ranging from page-locked memory to tensor cores on GPUs to MPI communication.

### A. Single-Particle Imaging (SPI) Reconstruction Algorithm

The ‘‘Multi-Tiered Iterative Phasing’’ (MTIP) algorithm [8] reconstructs 3D protein structures from X-ray diffraction data. It iteratively updates an electron density map of a macromolecule by making it consistent with real-space constraints and the data. In particular, the algorithm imposes the condition that the diffraction data derived from the current electron density map of the model matches the experimental diffraction data. The MTIP algorithm is modified from its original polar coordinates to Cartesian coordinates to make it parallelizable.

Fig. 2 shows the four components of the SpiniFEL workflow: (1) slicing, (2) orientation matching, (3) merging, and (4) phasing. In addition, the final reconstruction resolution is calculated post-analysis using the Fourier shell correlation metric [9] [10]. We describe each of the above steps in detail below using the notation specified in Fig. 2.

1) *Slicing*: Starting from an estimate of the real-space 3D **autocorrelation of electron density**  $A(\mathbf{r})$ , which is initially random, slicing creates a 3D volume of Fourier-space **model diffraction images**  $I(\mathbf{q}_k)$  (slices) at various orientations along the Ewald sphere. A predefined set of **reference orientations** is used. These 3D reference orientations are used to define the model. The more reference orientations there are, the better the model that can be constructed. The model diffraction images  $I(\mathbf{q}_k)$  are generated by applying the type-II (forward) non-uniform Fast Fourier Transform (NUFFT) on the estimated autocorrelation of electron density  $A(\mathbf{r})$  along the reference rotation orientations  $\mathbf{q}_k$ . Previous work in SpiniFEL used the `cufinufft` library [11] to efficiently calculate the forward (uniform to non-uniform) FFT on GPUs. This step scales at a rate of  $O(M)$ , where  $M$  is the number of model diffraction images  $I(\mathbf{q}_k)$ .

2) *Orientation Matching*: The orientation matching uses the pairwise Euclidean distance to compare the experimental diffraction images  $I_{\text{exp}}$  to the model diffraction images  $I(\mathbf{q}_k)$ . We then find the closest model diffraction image for every experimental diffraction image  $I_{\text{exp}}$  and the respective reference orientation that generated the model diffraction image  $I(\mathbf{q}_k)$ . The resulting orientations are  $\mathbf{q}_k^{\text{exp}}$ . This step scales as  $O(MN_{\text{exp}})$ , where  $M$  is number of model diffraction images and  $N_{\text{exp}}$  is number of experimental diffraction images.

Previous work for this step utilized a GPU-tailored, hand-optimized code for calculating the Euclidean distance. It then used a GPU implementation of the heap sort algorithm to find the minimum distance [4].

3) *Merging*: The merging step combines the 2D experimental diffraction data  $I_{\text{exp}}$  at the matched reference orientations  $\mathbf{q}_k^{\text{exp}}$ . This is accomplished by inverting the type-II NUFFT on oriented experimental data to solve for the real-space autocorrelation of the electron density estimate  $A^{\text{new}}(\mathbf{r})$ . Due to limitations in direct beam experiments, there are regions of missing data in the center corresponding to low-frequency information. This combined with the large amount of noise in the experiment makes solving this system ill-posed.

This issue is overcome by using Tikhonov regularization to approximate the exact solution, which transforms the problem into solving the following equation:

$$A^*DAx + \lambda(x - x_0) = A^*Db \quad (1)$$

where  $A$  is the type-II NUFFT operator,  $D$  are the weights,  $x$  is the vector representing the autocorrelation on a uniform grid,  $b$  is the vector of the intensity data on a uniform grid,  $x_0$  is the initial guess of the autocorrelation, and  $\lambda$  is the regularization parameter. This necessitates only computing one type-I NUFFT and then using the conjugated gradient (CG) to solve the linear system. We only need one type-I NUFFT to compute  $A^*Db$ .  $A^*DA$  has Toeplitz structure, meaning that it can be expressed as a discrete convolution, which can be computed in  $O(N^3 \log N)$  time using FFTs, where  $N$  is the grid size defined in Fig. 2.

4) *Phasing*: The phasing step converts the new 3D autocorrelation  $A^{\text{new}}(\mathbf{r})$  to a new 3D diffraction volume  $I^{\text{new}}(\mathbf{q})$  via

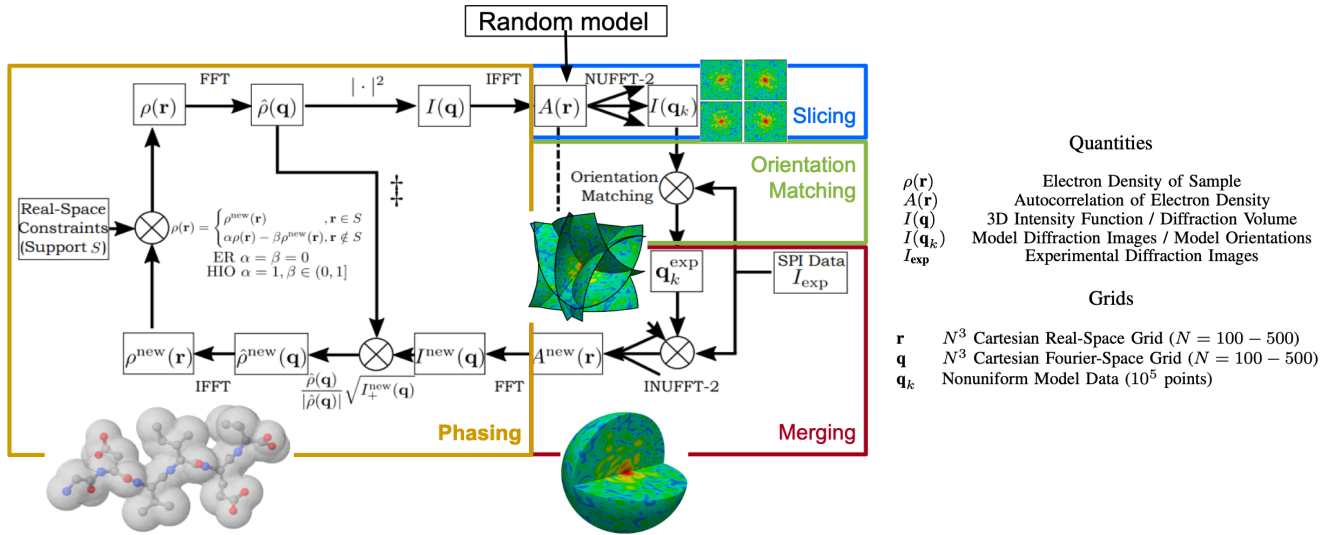


Fig. 2: MTIP algorithm and its notation for quantities and grids, as implemented within the SpiniFEL application. Image courtesy of ExaFEL project. Table from [4].

an FFT and iteratively converts it into a molecular structure by applying real-space and Fourier space constraints on the electron density  $\rho(\mathbf{r})$  of the sample using an iterative phasing algorithm. After phasing,  $I(\mathbf{q})$  is produced by taking the squared magnitude of the FFT of  $\rho(\mathbf{r})$ , and then  $A(\mathbf{r})$  is calculated by taking the inverse FFT of  $I(\mathbf{q})$ . The MTIP algorithm can then start again with  $A(\mathbf{r})$  at the slicing step.

5) *Fourier Shell Correlation (FSC)*: The FSC algorithm calculates the normalized cross-correlation coefficient between two three-dimensional (3D) volumes in Fourier space.

$$FSC(R) = \frac{\sum_{r_i \in R} F_1(r_i) F_2(r_i)^*}{\sqrt{\sum_{r_i \in R} |F_1(r_i)|^2 \sum_{r_i \in R} |F_2(r_i)|^2}} \quad (2)$$

where  $F_1$  and  $F_2$  are the volumes,  $*$  represents the complex conjugation operation, and  $R$  is a vector with spatial frequency  $|R|$ . SpiniFEL uses FSC to calculate the resolution between the generated model and the base model. This resolution is used to test whether the model has converged or not.

### B. Page-Locked Memory

Page-locked (pinned) memory is a region of memory where the operating system guarantees that the memory will *not* page out to disk. This allows the GPU to use direct memory access (DMA) to directly access the memory region, unlike pageable memory, where there would be an extra copy of data created while moving the data to the GPU memory. Using page-locked memory improves GPU bandwidth utilization, thus allowing for faster transfers [12].

### C. Tensor Cores on GPUs

NVIDIA introduced tensor cores with its Volta architecture for AI and HPC workloads. These tensor cores perform matrix multiplication significantly faster than their native cores. Users can directly access the core functionality of tensor cores with

### Algorithm 1 Symmetric All-to-All Exchange

---

**Input:** rank, comm\_size, buffer, batch\_size

```

for i ← 0 to comm_size do
  if i != rank then
    Send_data(buffer + i * batch_size, batch_size)
  end if
end for
for i ← 0 to comm_size do
  if i != rank then
    Recv_data(buffer + i * batch_size, batch_size)
  end if
end for

```

---

the CUDA WMMA API, or they can utilize NVIDIA libraries like cuBLAS that provide higher-level wrappers for BLAS GEMM [12]. Tensor cores operate at the warp level, where each warp performs matrix multiplication on a fragment of a supported size, with limitations on their supported data types.

### D. One-Sided Communication

One-sided communication, a form of remote memory access (RMA), utilizes the global address space to directly another process' memory. By decoupling data movement and process synchronization, we can efficiently transfer data between nodes without stalling.

One-sided communication in MPI works by creating memory windows that are accessible to all participating processes. Every process can read/write to this memory window without participation from the host process. Additionally, in the case of shared memory, processes can directly access the memory, bypassing the need to use a transfer buffer [13]. There are other libraries, such as GASNet-EX, that provide similar programming abstractions for RMA on distributed memory systems [14].

### E. Symmetric All-to-All Communication Paradigms

Symmetric all-to-all communication is a paradigm by which  $N$  processes can send their data to all the other processes. Using two-sided communication, the strategy could be implemented as shown in Algorithm 1.

There exist many algorithms that can improve the performance of this MPI all-to-all communication [15]. These algorithms primarily employ strategies to reduce overhead and minimize data movement and contention. Hofmann et al. [15] propose a linear shift algorithm that decreases the number of stalls and reduces the pressure on the network infrastructure by offsetting the order in which the memory of all processes is accessed [15].

## III. METHODOLOGY

Here we present our component-wise optimization of the SPI workflow to improve the overall workflow performance.

### A. Optimizing and Distributing Slicing

We discuss the components that we use to optimize the data movement of the model diffraction images  $I(\mathbf{q}_k)$  at their various orientations over the participating nodes. Hereafter, we refer to these as reference orientations or just *orientations*. These components distribute and optimize the access to non-local data in a way that minimizes contention and pipelining the transfer to minimize data transfer latency.

1) *Storing and Pinning Data Structures*: The baseline GPU implementation recomputed the reference orientations at every iteration of the algorithm. Part of this step is to use the `skopi` [16] package to generate random uniform quaternions and transform them into our reference rotation matrix. We reduce the computational cost of this step by performing the computation only once and storing the results in pinned memory to optimize the transfer to the GPU.

2) *Data Distribution*: We leverage memory windows, introduced in MPI 4.0 [13], to allow transparent and efficient access to data residing on non-local ranks. Every rank in a node allocates a shared memory window that can be accessed by the other ranks within the node; this buffer is then used to construct another window that is visible to all the other nodes, as shown in Fig. 3. In addition, we pin this memory to improve the data transfer of this buffer to the GPU as well as allow it to be pipelined.

Multiple ranks now each have a chunk of all the reference orientations. Ranks will need to use one-sided communication to copy non-locally resident orientations to local memory. Locally resident memory can be directly accessed and thus transferred to the GPU for processing.

We also note that using two-sided communication strategies for these types of problems is a bad fit, as they require participation from both the sender and receiver. Since multi-threading in Python has limitations due to the global interpreter lock (GIL), we identify one-sided communication as a much better fit for solving these types of problems. One-sided communication operates using RMA and can take advantage

of Infiniband [17]. We propose three strategies to define how much data to distribute over the nodes.

- *Completely Replicated Strategy*. This strategy replicates the orientations for each rank. It maximizes local memory access but comes at the cost of limiting the largest supported problem size.
- *Completely Distributed Strategy*. This strategy distributes all the reference orientations evenly among all the participating ranks. It minimizes memory utilization but comes at the cost of increased communication overhead.
- *Hybrid Replication Distribution Strategy*. This proposed hybrid strategy is a heuristic for distributing the data to maximize performance while also enabling large-scale science. We expose the *split size* hyperparameter that allows us our heuristic to tune for application performance for the architecture. The parameter defines the maximum optimal memory allocation size per node, and the heuristic uses this to find the data distribution between the nodes that maximally uses as much of this storage as possible while maximizing data replication. Thus, if the *split size* is less than the amount required to store all the orientations, then all data would be replicated across the nodes. Additionally, if *split size* is equal to  $\frac{\text{Required\_storage}}{\text{Num\_nodes}}$ , then it will lead to perfect distribution.

Different clusters have different memory storage and communication characteristics. Each cluster will have a different memory capacity wall, as shown in Fig. 4, where we cannot increase replication due to limited storage capacity. Decreasing replication will increase contention and may lead to degraded performance. Using the data distribution strategies, we can efficiently spread the data in a way that maximizes performance for the problem.

3) *Low-Contention Multi-Stream Access (LCMA) Strategy*: We initially implemented a trivial communication strategy where every node would access the data from the other ranks in order. Our initial naive access strategy, as shown in Fig. 5, labeled *Distributed*, performed 13% worse than when using perfect replication, labeled *Replicated*, because of communication overheads. Additionally, we expect these overheads to become even more prevalent as more nodes participate in the transfer. To overcome this bottleneck, we propose a novel heuristic strategy, LCMA, detailed in Algorithm 2, to perform our symmetric all-to-all communication. Hofmann et al. [15] designed a linear shift strategy that balances the accesses to the different nodes for a two-sided communication paradigm. We adapt this for one-sided communication and extend it to support multiple streams and multiple ranks per node, which is the case for most multi-GPU per-node setups. This strategy distributes work, as shown in Fig. 6, and offsets the access to the data to minimize the number of ranks that are accessing a single node's data. It is implemented as shown in Algorithm 2.



Fig. 3: Data distribution strategy. Illustration of how reference orientations and experimental images are distributed in the current strategy vs. the proposed strategy.

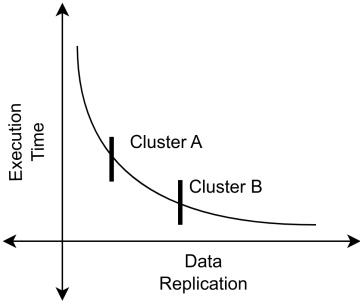


Fig. 4: Visual for balancing data replication with performance.

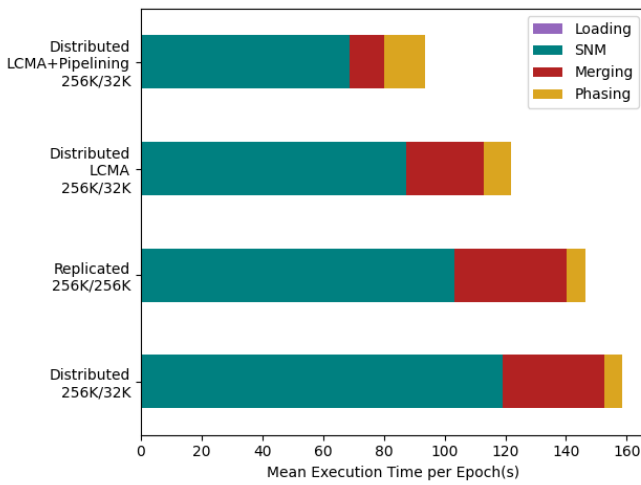


Fig. 5: Profiling information for the effect of data distribution and LCMA for 8 nodes and 2500 images per rank.

#### Algorithm 2 LCMA Algorithm

**Input:** shared\_comm\_size, comm\_size, rank, stream\_id, n\_streams, splits

**Output:** list of target ranks

**Require:** stream\_id < n\_streams, rank < comm\_size

**Ensure:** comm\_size mod shared\_comm\_size = 0

num\_nodes  $\leftarrow$   $\lfloor$  comm\_size / shared\_comm\_size  $\rfloor$

local\_rank  $\leftarrow$  rank mod shared\_comm\_size

node\_id  $\leftarrow$   $\lfloor$  rank / shared\_comm\_size  $\rfloor$

**Ensure:** num\_nodes mod splits = 0

num\_nodes\_in\_split  $\leftarrow$   $\lfloor$  num\_nodes / splits  $\rfloor$

offset\_node\_id  $\leftarrow$  num\_nodes mod splits

offset\_in\_split  $\leftarrow$   $\lfloor$  node\_id / num\_nodes\_in\_split  $\rfloor$

**for** i, idx  $\leftarrow$  enumerate(local\_rank to shared\_comm\_size, 0 to local\_rank) **do**

**for** j, jdx  $\leftarrow$  enumerate(offset\_node\_id to num\_nodes\_in\_split, 0 to offset\_node\_id) **do**

        target\_rank  $\leftarrow$  (i + j \* shared\_comm\_size) mod (shared\_comm\_size \* num\_nodes\_in\_split)

        stream\_id\_target  $\leftarrow$  (idx \* num\_nodes\_in\_split + jdx) mod n\_streams

**if** stream\_id = stream\_id\_target **then**

            target\_rank\_offset  $\leftarrow$  target\_rank + offset\_in\_split \* num\_nodes\_in\_split + shared\_comm\_size

**yield** target\_rank\_offset

**end if**

**end for**

**end for**

For a single stream, no two ranks will simultaneously access one rank's data. Additionally, we offset the order of access for ranks shared by a node. This acts as a stress release point, in case that (1) communication overheads continue to accumulate, (2) the pipelining step cannot keep up, or (3) a step of shared access is needed to allow the computing step to catch up again.

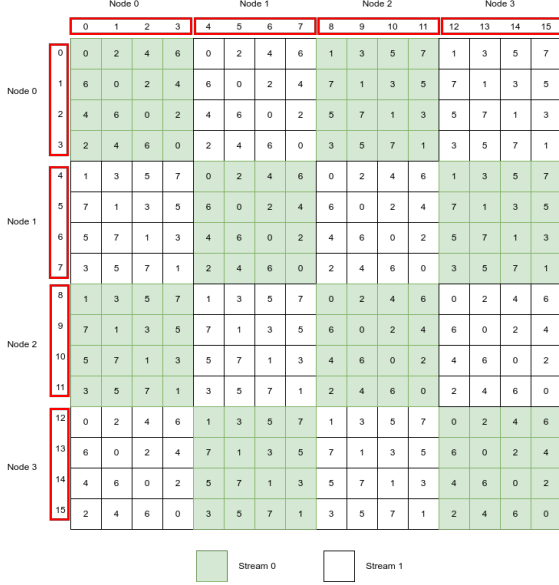


Fig. 6: Illustration of communication pattern for four nodes (16 ranks) with two streams. Each entry  $(i, j) \ni 0 \leq i, j < 16$  denotes the order in which the data in  $j$  is processed by  $i$  by the noted stream.

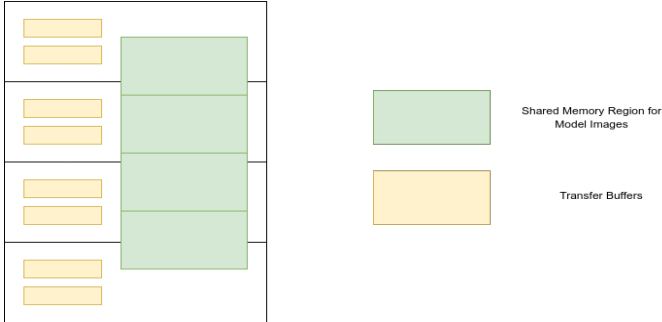


Fig. 7: Node layout. Illustration of how the different memory regions are shared between the ranks of the node.

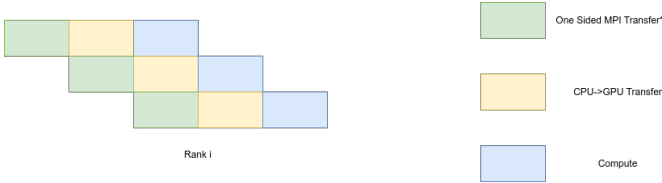


Fig. 8: Pipelined computation strategy. Illustration of how each rank pipelines the compute step with the transfers to hide the communication latency.

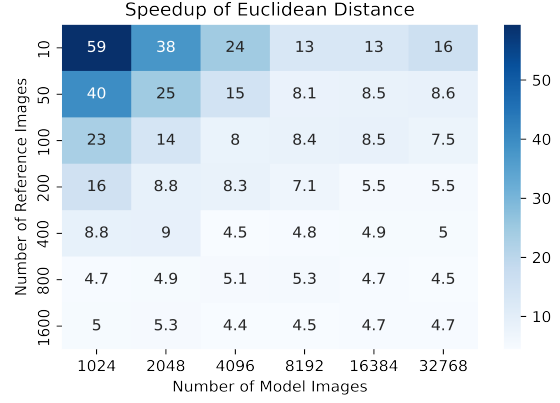


Fig. 9: Speedup vs. existing implementation for (1, 128, 128) sized images.

4) *Pipelining Computation*: In addition to improving our access pattern, we further optimize the transfer by pipelining the operations. We construct  $N$  pinned transfer buffers, which correspond to the number of streams that we use, to transfer non-local data to be accessed locally, as shown in Fig. 7. Since all the batches are independent, we can use multiple streams over multiple threads to pipeline compute operations with the data transfer and, in turn, help hide the transfer overhead, as shown in Fig. 8.

### B. Acceleration of Orientation Matching

We propose a novel algorithm for comparing the experimental images with model images by using the pairwise Euclidean distance. We take advantage of specialized matrix cores available on modern GPUs to significantly speed up our problem by changing our formulation of the problem. This formulation makes our implementation both portable and much more efficient.

We write the Euclidean distance of two vectors as

$$(\vec{x} - \vec{y})^2 = \|\vec{x}\|^2 + \|\vec{y}\|^2 - 2 \langle \vec{x}, \vec{y} \rangle \quad (3)$$

When we extend this to tensors, we can write it as

$$\|(X_{A,n} - Y_{B,n})\|^2 = \sum_{i=1}^n X_{A,i}^2 \oplus \sum_{i=1}^n Y_{B,i}^2 - 2X_{A,n} \cdot Y_{B,n}^T \quad (4)$$

The outer product is computed by broadcasting the squared norm of the images. The matrix product can be computed using the GEMM BLAS routine  $C := C - 2 * X * Y^T$  on the GPU. By using this routine to calculate the matrix product, we can significantly accelerate the computation by using matrix cores available on modern GPUs.

### C. Finding the Closest Reference Orientations

The previous implementation used a GPU version of heap sort to find the row-wise minimum. We take advantage of CuPy's `argmin`, which internally uses NVIDIA's `cupy::DeviceSegmentedReduce::ArgMin` [18]. This uses a GPU optimized tree-style reduction algorithm to find the index of the minimum.

#### D. Batching the Argmin Step

When processing the orientations, to reduce the memory footprint for storing the partial minimums, we design and implement a batched `argmin` implementation, as shown in Algorithm 3.

---

**Algorithm 3** Batched Argmin

---

```
dist  $\leftarrow$  [batch_size + 1, N_slices]
args  $\leftarrow$  [N_slices]
for target_rank  $\leftarrow$  0 to ranks do
  for offset  $\leftarrow$  0 to batch_size do
    dist[:1]  $\leftarrow$  PairwiseEuclideanDist(model_images,
reference_images[target_rank])
    a_t  $\leftarrow$  ArgMin(dist, axis=0)
    m_i  $\leftarrow$  a_t  $\neq$  batch_size
    args[m_i] = target_rank * N_images_per_rank +
offset * batch_size + a_t[m_i]
    md  $\leftarrow$  TakeAlongAxis(dist, a_t, 0)
    dist[-1]  $\leftarrow$  md
  end for
end for
```

---

This strategy stores the minimum distances in the last column of the distance matrix *dist* and stores the index of the minimums in a separate *args* array. For every batch, we update the distance matrix for  $[0, \text{batch\_size})$  and update the *arg* array for values that are not located in the last column of the *dist* matrix. This strategy allows us to use just one Euclidean distance calculation and *argmin* calculation per batch.

#### E. Accelerating Merging

Previous work for SpiniFEL [19] identified low utilization for this step. Further investigation identified that the merging step not only involved significant data movement between the CPU and GPU but also many memory allocations and de-allocations. To remedy this, we identified redundant computations between iterations, pre-computed their results, and stored them in memory. In addition, we store the plans for the NUFFTs and arrays for intermediate calculation to reduce fragmentation and allocations/de-allocations. We further improve the performance of merging by moving the CG calculation to the GPU using CuPy [20].

We also found that the CG algorithm used in our merging step is sensitive to small perturbations [10]. Utilizing double precision improves convergence. To deliver both high performance and stability in the CG algorithm, we added support for mixed-precision arithmetic. (For testing, we used double precision to ensure stability.)

#### F. Accelerating the Resolution Computation

We identify computationally intensive operations in the Fourier shell correlation (FSC) calculation and start processing those components on the GPU. We use CuPy’s accelerated kernels to parallelize FSC computation. This achieves a  $40\times$  speedup when the computation is done outside SpiniFEL and

a  $485\times$  speedup when done within the SpiniFEL iteration, storing intermediates and preloading libraries.

## IV. RESULTS AND DISCUSSION

We run a variety of scaling studies on the GPU nodes of the NERSC Perlmutter [21] supercomputer, each of which has a third-generation AMD EPYC 7763 CPU and four NVIDIA A100 GPUs. It uses the Slingshot 11 interconnect for internode communication. All of our tests have four ranks per node, each with exclusive access to one GPU.

For all of our tests, we use the 3IYF dataset, which has an image size of (1,128,128). Each test runs identical iterations of the entire workflow, until the model converges, with a maximum of 10 iterations. We present the total execution time for the compute optimizations but the average execution time for the distributed runs, as some of those runs stopped iterating once they reached their convergence thresholds.

#### A. Assessing the Effectiveness of the Compute Optimizations

These experiments compare the improvement of SpiniFEL when all the orientations fit inside memory and thus there is no communication overhead for transferring the orientations across the nodes. We compare the performance gains in *Slicing*, *Orientation Matching*, and *Merging* and the additional overhead in *Loading* between the baseline and our optimized version. We also note that we associate performance differences to *Phasing* as run-to-run variance, as we made no changes to that component of the workflow.

1) *Improvements to Pairwise Euclidean Distance*: Fig. 9 shows the speedup achieved over the existing hand-optimized version for a range of model images and experimental images for images of size (1, 128, 128), performed on an NVIDIA A100 GPU.

We see significant gains in performance for all combinations of the number of experimental and model images. Additionally, we can use high-level primitives to implement this component, which in turn enables us to easily port this to other architectures and achieve high performance.

2) *Weak Scaling Study*: We perform a weak scaling study by fixing the number of images per rank and the number of reference orientations used, varying the number of nodes, and measuring the final execution time, as shown in Fig. 10 and Fig. 12.

From Fig. 10 we can see that we get a  $2.84\times$ ,  $118.79\times$ , and  $7.59\times$  speedup for *Slicing*, *Orientation Matching*, and *Merging* respectively, and a  $0.5\times$  speedup for *Loading*. Combined, our optimizations net an overall average performance gain of  $5.28\times$  when varying the number of nodes. Additionally, from Fig. 12, we see that our implementation does not suffer from communication bottlenecks in the merging or phasing steps. We also note that compute optimizations do not experience performance regression as we increase the number of nodes.

3) *Performance Characterization of SpiniFEL when Varying the Number of Orientations*: We study the effect of increasing the number of orientations, ensuring that all the

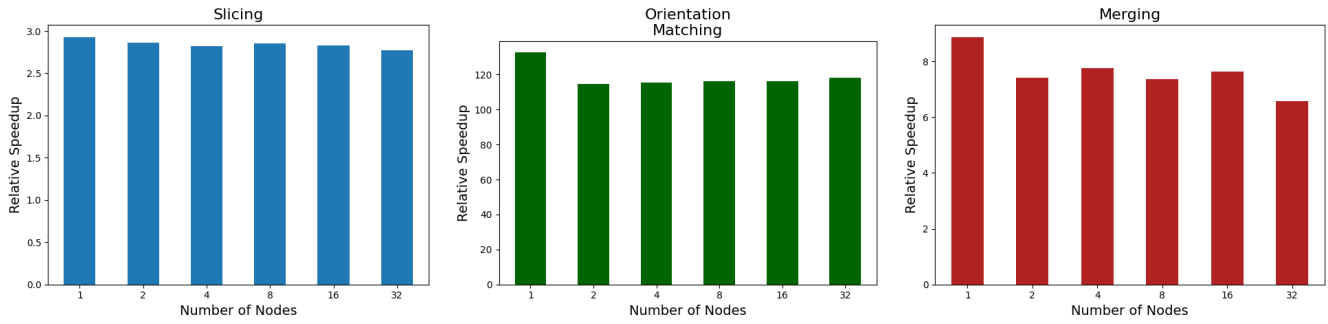


Fig. 10: Comparing relative speedup for individual components of workflow between baseline GPU and optimized GPU when changing the number of nodes: 1500 images per rank and 10K orientations.

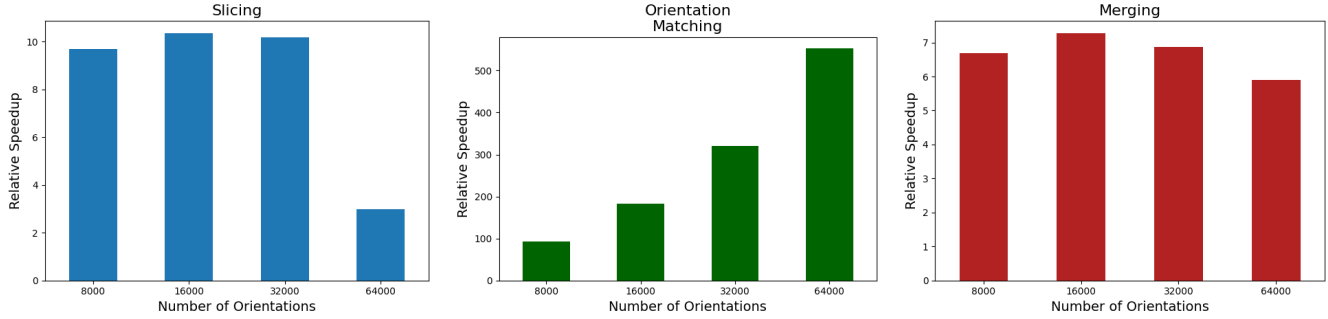


Fig. 11: Comparing relative speedup for individual components of workflow between baseline GPU and optimized GPU when changing the number of orientations using 32 nodes and 1500 images per rank.

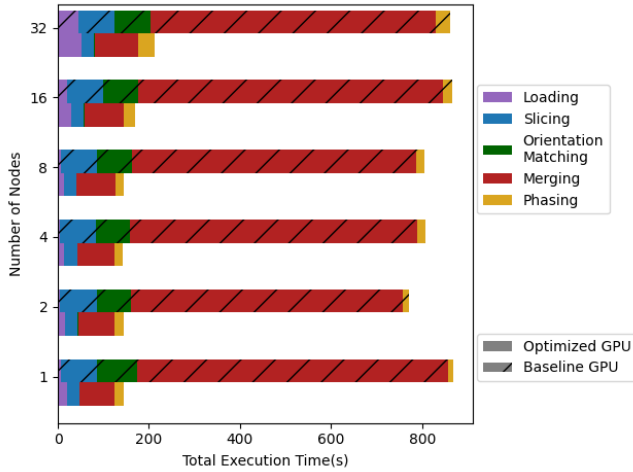


Fig. 12: Comparing total execution time of Fig. 10.

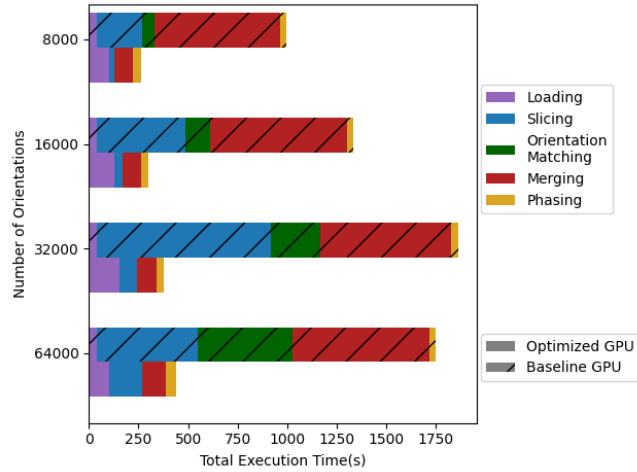


Fig. 13: Comparing total execution time of Fig. 11.

orientations fit in memory and keeping the number of nodes and images per rank the same, as shown in Fig. 11 and Fig. 13.

Fig. 11 shows a  $8.3\times$ ,  $287.22\times$ , and  $6.31\times$  speedup for *Slicing*, *Orientation Matching*, and *Merging*, respectively, and a  $0.33\times$  speedup for *Loading*. Combined, these optimizations deliver an overall average performance gain of  $4.19\times$  when varying the number of orientations.

Fig. 13 shows that the execution time of SpiniFEL increases linearly as we increase the number of orientations, except for 64000 orientations. In-depth profiling of the application reveals that the `skopi::get_uniform_quat()` function shows ir-

regular behavior with the number of orientations. Interestingly, even `cufinufft` performs the forward uniform to non-uniform FFT faster for these cases. We see this behavior in more depth in Fig. 14.

Since our optimized version only performs this calculation once, during *Loading*, we see larger speedups for the more time-consuming cases. Thus, SpiniFEL will perform better for certain orientation configurations, but its effect on the convergence of the problem still needs to be studied. We plan to explore this behavior of `Skopi` and `cufinufft` in more depth in the future.

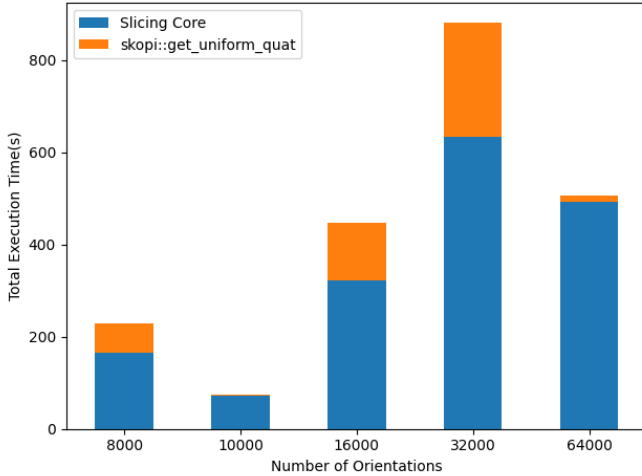


Fig. 14: Irregular execution time of slicing when varying the number of orientations with `skopi::get_uniform_quat()` and `cufinufft`.

We thus see that our optimized version scales linearly as we increase the number of orientations. By storing the reference rotation matrix, we ensure the application performance is not bottlenecked by the `skopi::get_uniform_quat()` function.

### B. Assessing the Efficiency of LCMA

These experiments show the behavior of SpiniFEL for cases when the number of orientations is too large to fit into memory. Thus, communication with the LCMA is needed for sharing the reference orientations between the nodes. Since the out-of-core execution strategies closely tie *Slicing* and *Orientation Matching*, we refer to the overall computation step as *SNM*. We conduct these tests using 2500 images per rank.

We saw the effect of LCMA and pipelining in Fig. 5 as Distributed Optimized, where there was a 20% speedup even over replication. Using MPI communication alleviates a memory access bottleneck that arises when all four of the ranks per node simultaneously access the memory.

1) *Weak Scaling Study with Complete Distribution*: We study the *Completely Distributed* strategy to spread the orientations over all the nodes by keeping the number of images per rank fixed at 2500 images-per-rank and linearly increasing the number of orientations with the number of nodes used. This is shown in Fig. 15.

We see that the execution time for *SNM* increases linearly for at least 32 nodes. Thus, we see that the effect of communication is negligible.

2) *Fixed Scaling Study when Using Hybrid-Distribution Replication*: We study the *Hybrid Distributed Replicated* strategy to spread the orientations across nodes in a way that minimizes communication. We keep the number of images per rank fixed at 2500 images-per-rank, the *split size* fixed at 256K, and test for 512K and 1024K orientations, which the results can be seen in Fig. 16. We see that as we go from 512K to 1024K orientations, the execution time doubles, and

as we increase the number of nodes, the increase in execution time for processing the orientations is negligible. This shows that our strategy works, as expected.

3) *Assessing the Improvement to LCMA with Pipelining*: We study the effect of combining *LCMA* and *Pipelining* strategies on SpiniFEL. We fix the number of the orientations to 1024K Orientations, 2500 images-per-rank, and *split size* at 256K and evaluate the effectiveness of using two streams, as shown in Fig. 17. We see a  $1.2\times$  overall improvement in the *SNM* step. Additionally, we also see a  $2.34\times$  improvement in the average *Merging* time. The *Merging* step has a global communication barrier, and thus it waits for other ranks to reach that step. Pipelining not only improves the average execution time of *SNM* but also significantly reduces its variation. This reduces the time that stalled ranks have to wait for and thus improves the performance of the *Merging* step.

### C. Exploring the Hyperparameter Space of SpiniFEL

The formulation of SpiniFEL introduces several hyperparameters that enable us to tune the application to the target hardware and problem. We perform a study to find the best set of hyperparameters for our current test system, Perlmutter. We identify optimal parameter configurations by iterating over a pruned search space. The search time for the hyperparameters is equal to number the of proposals times the total execution time of each proposal.

1) *Effect of Batch Size Hyperparameter*: We study the effect of *batch size* hyperparameter for SpiniFEL. *Batch size* controls the granularity at which the orientations are processed with the experimental images. The optimal parameter will simultaneously maximize the network transfer rate with LCMA and Pipelining, and the rate at which the GPU can process the images. We see the results of this study in Fig. 18. Originally, a *batch size* of 100 was identified as optimal for the NVIDIA V100s with the Summit Supercomputer. We see a  $1.2\times$  performance gain as we increase the *batch size* from 100 to 1000 for the NVIDIA A100s. We also see a further  $1.04\times$  speedup as we increase the *batch size* from 1000 to 4000. However, this comes at the cost of increased memory consumption, and we note that our tests for using a batch size of 8000 crashed due to out-of-memory errors. Thus, we find that future systems with more onboard GPU memory and more memory bandwidth have a possible avenue to improve performance by using a larger batch size.

2) *Effect of the Number of Streams*: We study the effect of *Number of Streams* hyperparameter for SpiniFEL. *Number of Streams* controls the number of simultaneous threads that are processing the orientations. By running multiple streams, we can hide both the transfer time between nodes and the transfer time from CPU memory to GPU memory. This allows SpiniFEL to constantly feed the GPU with work and maximize performance. We see this in Fig. 19. We see a  $1.2\times$  and  $2.3\times$  performance improvement to *SNM* and *Merging*, and an overall speedup of  $1.25\times$  when we use 2 streams over using 1 stream. We also see that we get no improvement in performance when using more than 2 streams. This suggests

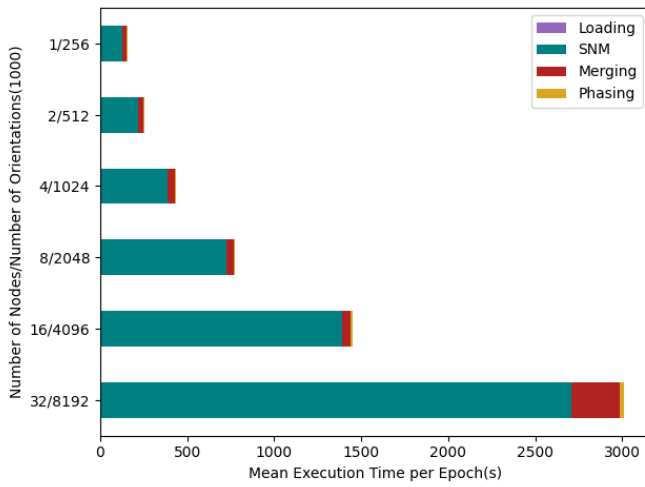


Fig. 15: Studying the scaling behavior of SpiniFEL(out-of-core) as we linearly increase the nodes and orientations.

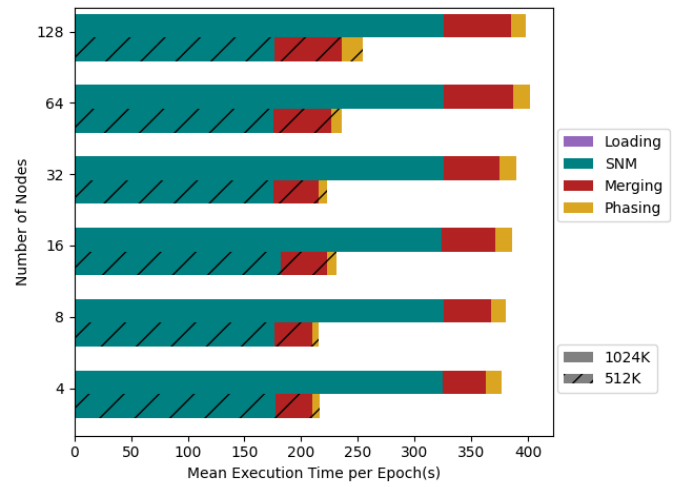


Fig. 16: Weak scaling of SpiniFEL(out-of-core) with 512K and 1024K orientations with a split size of 256K orientations.

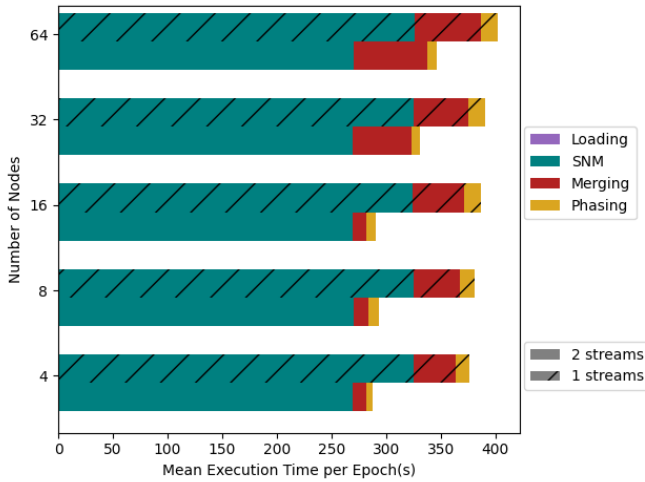


Fig. 17: Assessing the performance gains of using pipelining in SpiniFEL(out-of-core) with 1024K orientations and split size 256K orientations.

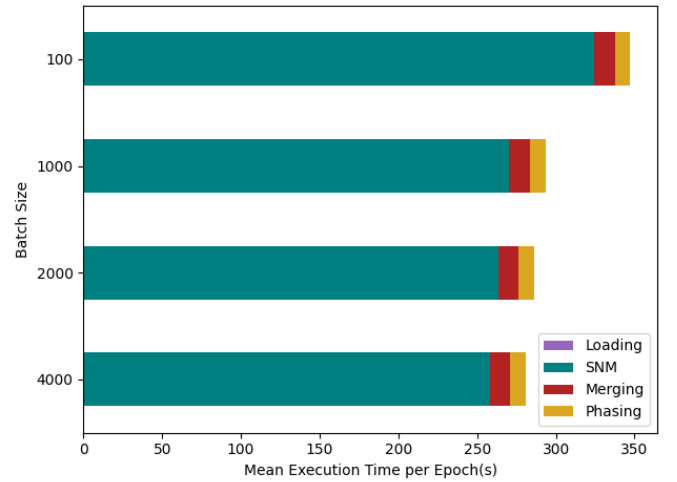


Fig. 18: Identifying the optimal batch size for SpiniFEL(out-of-core) using 1024K orientations and a split size of 256K orientations.

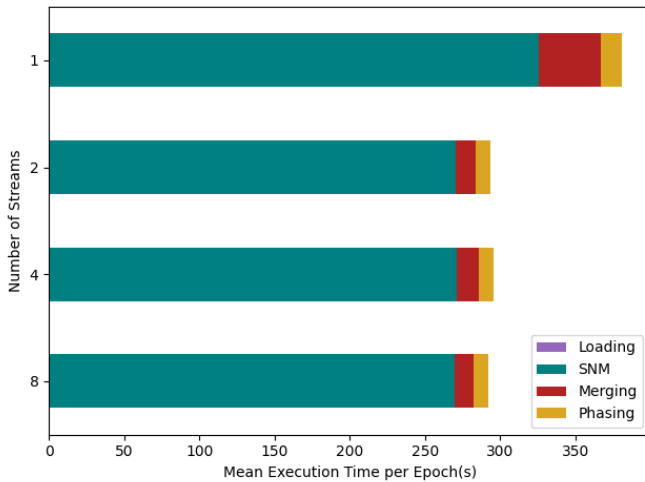


Fig. 19: Identifying the optimal number of streams in SpiniFEL(out-of-core) using 1024K orientations and split size of 256K orientations.

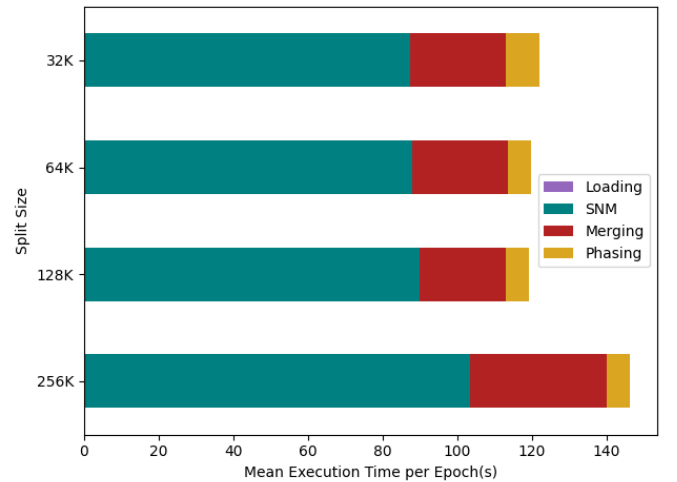


Fig. 20: Studying the effect of different split sizes for Hybrid Replication Duplication using 8 nodes and 256K orientations.

that at least for our current architecture, we are compute-bound for the components, and hiding the memory latency doesn't improve the performance. When combined with our previous  $5.28\times$  speedup over baseline, we get up to  $6.6\times$  speedup over baseline.

3) *Effect of the Split Size*: We study the effect of the *split size* hyperparameter on the *Hybrid Distributed Replicated* strategy. We keep the number of images per rank fixed at 2500 images per rank and the number of orientations set at 256K. Since for each of the runs, the amount of work done is the same, we can use this to understand the effect of communication when distributing the data. We see this Fig. 20. We see that as go from complete replication to distributing the data, we see a 15% performance uplift. We again attribute this to putting too much stress on the memory controller. Furthermore, we see that even as we reduce the data locality by decreasing the *split size*, the execution time for *SNM* remains consistent. We can thus infer from these results that our application scales strongly with the *split size* hyperparameter and that there are potential performance gains when keeping the data non-local.

## V. FUTURE WORK

### A. Jagged Data Distribution

The *Hybrid Replication Distribution* strategy minimizes the number of nodes required to share all the orientations, but it doesn't maximize local data presence. Future work will investigate a strategy where nodes that are sharing the data will have replication between them, to maximize the amount of data stored locally. This strategy will require nodes to store more metadata regarding what extra orientations it is storing.

### B. Supporting Shared Memory on GPUs

The current design still keeps all orientations on system memory and transfers them to the GPU when needed. We can retool the current design to keep all the data on GPUs, not system memory. Using technologies like NVSHMEM/OpenSHMEM, we can remove the intermediate step of using a transfer buffer when transferring data between nodes and the GPU. This would significantly reduce communication overheads and even overcome the memory bandwidth bottleneck that we found when all the data is local since every rank has a GPU with its own memory controller. There might be more communication operations over the network as GPU memory is usually smaller than system memory, and since ranks cannot pool GPU memory, there might be more latency when transferring data between local ranks. However, we predict that with LCMA, we can hide this latency.

### C. Understanding Irregular Behavior of *Skopi* and *cufinufft*

We found irregular behavior in *skopi::get\_uniform\_quat()* and *cufinufft* when we changed the number of orientations. We plan to investigate this behavior in more depth. Future research will be directed towards improving the understanding of the relationship between the number of orientations, convergence, and performance.

### D. Using Mixed Precision for Components

The current experiment uses double-precision arithmetic to ensure correct behavior. However, certain steps in the algorithm may not be as sensitive to precision as others, and mixing precision might enable us to improve performance without compromising on the accuracy.

## VI. SUMMARY

We achieve significant performance gains by profiling applications to guide the development of algorithms tailored to GPUs. Specifically, we can identify computational bottlenecks, and by using the strategies of pipelining computation, maintaining data locality, and using specialized hardware, we can maximize our application performance. Additionally, we show that by utilizing RMA in MPI, we can efficiently distribute data over multiple nodes. Furthermore, by utilizing optimized algorithms to effectively transfer our data between the nodes, we can minimize the contention of network and memory resources. We show in our results that our data distribution and access strategy are not bottlenecked by communication overheads when increasing the number of participating nodes. Simultaneously, by designing our algorithms to utilize hyperparameters, we can control data locality, data movement, parallelism, and model convergence. This flexibility enables future application developers to quickly tune the application for future architectures. Our work in improving the performance and scalability of SpiniFEL enables scientists to get results of more complex experiments in realtime.

## ACKNOWLEDGMENTS

We acknowledge the help of fellow ExaFEL team members at SLAC National Accelerator Laboratory: Chun Hong Yoon for his technical review of and supportive comments on the GPU acceleration work, Monarin Uervirojnangkoorn, Ariana Peck, Elliott Slaughter, and Seema Mirchandaney for help with testing and merging the GPU acceleration code, and Amedeo Perazzo for leading the ExaFEL project. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the US Department of Energy under Contract DE-AC02-05CH11231 using NERSC award DDR-ERCAP0029334. Los Alamos National Laboratory (LANL) is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). LA-UR-23-31392. This work was partially supported by the Center of Advanced Mathematics for Energy Research Applications, funded by the US Department of Energy's Office of Advanced and Scientific Computing Research and Basic Energy Sciences, under Contract DE-AC02-05CH11231.

## REFERENCES

- [1] P. Messina, "The Exascale Computing Project," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

- [2] C. Bostedt, S. Boutet, D. M. Fritz, Z. Huang, H. J. Lee, H. T. Lemke, A. Robert, W. F. Schlotter, J. J. Turner, and G. J. Williams, "Linac Coherent Light Source: The first five years," *Rev. Mod. Phys.*, vol. 88, p. 015007, Mar 2016. [Online]. Available: <https://link.aps.org/doi/10.1103/RevModPhys.88.015007>
- [3] J. Galayda, "The LCLS-II: A high power upgrade to the LCLS," *Proceedings of the 9th Int. Particle Accelerator Conf.*, vol. IPAC2018, 2018.
- [4] H.-Y. Chang, E. Slaughter, S. Mirchandaney, J. Donatelli, and C. H. Yoon, "Scaling and acceleration of three-dimensional structure determination for single-particle imaging experiments with SpiniFEL," 2021.
- [5] J. Blaschke, S. Mirchandaney, C. Yoon, E. Slaughter, M. Uervirojnangkoon, I. Chang, A. Dujardin, P. Kommera, V. B. Ramakrishnaiah, and C. Sweeney, "MTIP single particle imaging (spinifel) v0.1.0," 10 2021. [Online]. Available: <https://www.osti.gov/servlets/purl/1834376>
- [6] R. Neutze, R. Wouts, D. van der Spoel, E. Weckert, and J. Hajdu, "Potential for biomolecular imaging with femtosecond X-ray pulses," *Nature*, vol. 406, pp. 752–7, 09 2000.
- [7] T. D. Goddard, C. C. Huang, E. C. Meng, E. F. Pettersen, G. S. Couch, J. H. Morris, and T. E. Ferrin, "UCSF ChimeraX: Meeting modern challenges in visualization and analysis," *Protein Science*, vol. 27, no. 1, pp. 14–25, 2018.
- [8] J. J. Donatelli, J. A. Sethian, and P. H. Zwart, "Reconstruction from limited single-particle diffraction data via simultaneous determination of state, orientation, intensity, and phase," *Proceedings of the National Academy of Sciences*, vol. 114, no. 28, pp. 7222–7227, 2017. [Online]. Available: <https://www.pnas.org/doi/abs/10.1073/pnas.1708217114>
- [9] G. Harauz and M. van Heel, "Exact filters for general geometry three dimensional reconstruction," *Optik*, vol. 73, no. 4, pp. 146–156, 1986.
- [10] W. W. Hager and H. Zhang, "A survey of nonlinear conjugate gradient methods," *Pacific journal of Optimization*, vol. 2, no. 1, pp. 35–58, 2006.
- [11] Y.-h. Shih, G. Wright, J. Andén, J. Blaschke, and A. H. Barnett, "cuFIN-UFFT: a load-balanced GPU library for general-purpose nonuniform FFTs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 688–697.
- [12] NVIDIA, *CUDA C++ Programming Guide Release 12.2*, 2023. [Online]. Available: [https://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf)
- [13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- [14] D. Bonachea and P. H. Hargrove, "GASNet-EX: A high-performance, portable communication library for exascale," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, ser. Lecture Notes in Computer Science, vol. 11882. Springer International Publishing, October 2018, Lawrence Berkeley National Laboratory Technical Report (LBNL-2001174). [Online]. Available: <https://www.springer.com/us/book/9783030346263>
- [15] M. Hofmann and G. Runger, "In-place algorithms for the symmetric all-to-all exchange with MPI," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 105–110. [Online]. Available: <https://doi.org/10.1145/2488551.2488568>
- [16] A. Peck, H.-Y. Chang, A. Dujardin, D. Ramalingam, M. Uervirojnangkoon, Z. Wang, A. Mancuso, F. Poitevin, and C. H. Yoon, "Skopi: a simulation package for diffractive imaging of noncrystalline biomolecules," *Journal of Applied Crystallography (Online)*, vol. 55, no. 4, 7 2022.
- [17] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and O. Panda, "Host-assisted zero-copy remote memory access communication on InfiniBand," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 31–.
- [18] P. V. NVIDIA and F. H. Fitzek, "CUDA, release: 10.2. 89," 2020.
- [19] P. R. Kommera, V. Ramakrishnaiah, C. Sweeney, J. Donatelli, and P. H. Zwart, "GPU-accelerated multitiered iterative phasing algorithm for fluctuation X-ray scattering," *J Appl Crystallogr*, vol. 54, no. Pt 4, pp. 1179–1188, Aug 2021.
- [20] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: [http://learningsys.org/nips17/assets/papers/paper\\_16.pdf](http://learningsys.org/nips17/assets/papers/paper_16.pdf)
- [21] NERSC, "PerlMutter Architecture," <https://docs.nersc.gov/systems/perlmutter/architecture/>.