# Large-scale Nanostructure Simulations from X-ray Scattering Data On Graphics Processor Clusters

Abhinav Sarje
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720
Jack Pien
Consultant, 1216 North Road, Belmont, CA 94002
Xiaoye S. Li
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720
Elaine Chan, Slim Chourou, Alexander Hexemer
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720
Arthur Scholz and Edward Kramer
University of California, Santa Barbara, CA 93106

**Abstract**

X-ray scattering is a valuable tool for measuring the structural properties of materials used in the design and fabrication of energy-relevant nanodevices (e.g., photovoltaic, energy storage, battery, fuel, and carbon capture and sequestration devices) that are key to the reduction of carbon emissions. Although today's ultra-fast X-ray scattering detectors can provide tremendous information on the structural properties of materials, a primary challenge remains in the analyses of the resulting data. We are developing novel high-performance computing algorithms, codes, and software tools for the analyses of X-ray scattering data. In this paper we describe two such HPC algorithm advances. Firstly, we have implemented a flexible and highly efficient Grazing Incidence Small Angle Scattering (GISAXS) simulation code based on the Distorted Wave Born Approximation (DWBA) theory with C++/CUDA/MPI on a cluster of GPUs. Our code can compute the scattered light intensity from any given sample in all directions of space; thus allowing full construction of the GISAXS pattern. Preliminary tests on a single GPU show speedups over 125x compared to the sequential code, and almost linear speedup when executing across a GPU cluster with 42 nodes, resulting in an additional 40x speedup compared to using one GPU node. Secondly, for the structural fitting problems in inverse modeling, we have implemented a Reverse Monte Carlo simulation algorithm with C++/CUDA using one GPU. Since there are large numbers of parameters for fitting in the in X-ray scattering simulation model, the earlier single CPU code required

weeks of runtime. Deploying the AccelerEyes Jacket/Matlab wrapper to use GPU gave around 100x speedup over the pure CPU code. Our further C++/CUDA optimization delivered an additional 9x speedup.

# Disclaimer



# 1 Introduction

The Advanced Light Source (ALS) located at the Lawrence Berkeley National Laboratory is a third-generation synchrotron light source, one of the world's brightest sources of ultraviolet and soft X-ray beams. It is a national user facility funded by the U.S. Department of Energy, and is internationally recognized for its world-class measurement capabilities in X-ray science. X-ray scattering is a valuable tool for measuring the structural properties of materials used in the design and fabrication of energy-relevant nanodevices (e.g., photovoltaic, energy storage, battery, fuel, and carbon capture and sequestration devices) that are key to the reduction of carbon emissions. These techniques permit characterization of material structures on length scales ranging from the sub-nanometer to microns and down to the millisecond time scale. For example, small angle X-ray scattering (SAXS) and grazing incidence SAXS (GISAXS) methods permit characterization of nanoscopic and near-surface structural features, respectively, that arise from the self-assembly of block copolymers into ordered microphases or the self-assembly of nanoparticles.

Figure 1 illustrates the GISAXS scattering geometry. The incident X-ray wave vector $\mathbf{k_i}$ is kept at a small grazing angle with respect to the sample surface to enhance the near-surface scattering. The scattered beam, of wave vector $\mathbf{k_f}$, makes the out-of-plane scattering angle $\alpha_\mathbf{f}$ with respect to the sample surface and the in-plane angle $\mathbf{2}\theta_\mathbf{f}$ with respect to the transmitted beam. For GISAXS, a 2D detector is used to record the intensity of the scattered wave vector. The measured intensity is a function of the angular coordinates $\alpha_\mathbf{i}$, $\alpha_\mathbf{f}$ and $\mathbf{2}\theta_\mathbf{f}$. The

incident angle $\alpha_i$ can be varied and the sample can be rotated by an angle $\omega$ around its surface normal, thus creating many 2D images with different various intensity profiles. Analysis algorithms are used to analyze these images and predict the atomic structure of the underlying sample being probed.

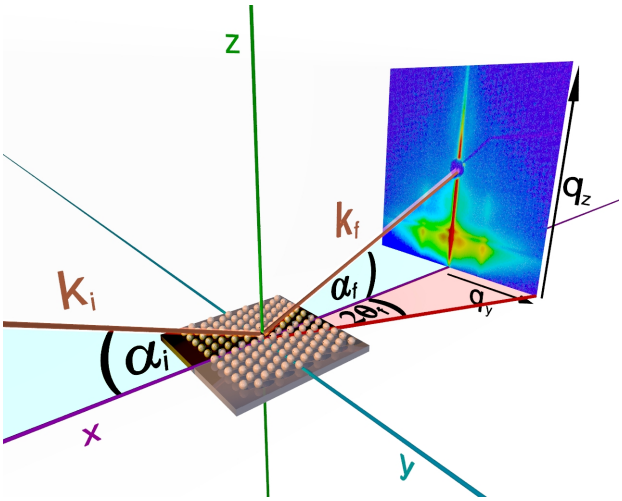


Figure 1: Grazing incidence small angle X-ray scattering (GISAXS) geometry. *Image taken from A. Meyer's www.gisaxs.de*

Although the scattering techniques described above can provide tremendous information on the structural properties of materials comprising nanoscale devices for energy technologies, a primary challenge remains in the analyses of the resulting data. An understanding of the fundamental physics that underlie the scattering methods is necessary to create accurate models and simulation algorithms for extracting information on material structures from the measured scattering patterns. Model and simulation development typically requires the development and implementation of computational hardware and software tools. Currently, the bottleneck in data analysis is the computational time required to complete the analysis, which is commonly of the order of several weeks to several months. The analysis time is compounded by the fast measurement rates of current state-of-the-art high-speed detectors. For example, users at Stanford's Linac Coherent Light Source (LCLS) facility can collect 24 terabytes of data in two weeks using a detector that outputs 100 megabytes of data per second. Quantitatively analyzing such massive sets of data in an intelligent and coherent manner is a daunting task at present. It is envisioned that such fast data collection rates will also be attainable at LBNL's future Next Generation Light Source (NGLS) facility. With the arrival of new state-of-the-art single photon counting detectors at the ALS that enable millisecond SAXS and $\mu$XRD measurements, the accumulation of large amounts of data, while necessary to develop a quantitative understanding of materials, poses a severe impediment

in designing a sequential set of studies. Consequently, the beamline scientists and the users are faced with an extremely inefficient utilization of the current light sources and recently developed detection systems. This mismatch must be removed before we can envision or effectively use any newly developed scattering beamline hardware.

We are developing new high performance computing algorithms, codes, and software tools for the analysis of X-ray scattering data collected at the ALS. The targeted parallel platforms are the large-scale parallel manycore systems with hybrid node architectures, including GPU accelerators.

General purpose graphics processors offer fine grained parallelism, where each core executes a lightweight thread according to the SIMD model [4]. Threads are scheduled to a multiprocessor on the GPU as thread blocks, while executed as warps, where one warp consists of 32 threads in the same thread block. Threads in a thread block can synchronize through the shared memory available on each multiprocessor. All thread blocks work asynchronously, and can only synchronize through the device memory. Because of this GPUs perform best on data parallel computations [1]. The less synchronization is needed, the better the performance would be. A number of programming models have been developed to harness the computational power of GPUs. High-level examples include AccelerEyes Jacket/Matlab which provides an interface to GPUs through Matlab. For higher-performance, lower level models are used. One example is Nvidia CUDA framework [5]. One uses C/C++ programming language to interface with GPUs through primitives available in CUDA.

In this paper, we present our recent results of the high performance implementation of the two most important classes of the analysis algorithms used in the X-ray scattering community. The first one is Distorted Wave Born Approximation (DWBA) model involving form factor computations, and the second is a more general Reverse Monte Carlo modeling approach which is usually slower than the DWBA method.

# 2 DWBA method

Grazing incidence small angle X-ray scattering (GISAXS) is a unique technique for investigating material topology and the structure of collections of nanoobjetcts deposited on top of surfaces or confined inside multilayered films. Simultaneous scanning of the in-plane and out-of-plane directions of the sample makes GISAXS as a comprehensive tool that produces images exhibiting detailed features of the underlying nanostructures, hence allowing a wealth of information compared to alternative methods. To date, the only theoretical framework to model the GISAXS process is the Distorted Wave Born Approximation (DWBA) method based on the perturbative solution of the electromagnetic wave propagation equation inside a stratified medium [6]. To our knowledge, only a handful of computer codes implementing the DWBA formalism have been made available for the community. Those codes are not sufficiently general to study the materials with complex structures, nor do they utilize the state-of-the-art high-

performance computer systems.

Studying highly complex structures – which is the main objective of GISAXS – requires solving for the **form factor** in a high-resolution **k**-space grid, resulting typically in matrices with tens to hundreds of million points. This time-consuming and memory-demanding calculation constitutes a major bottleneck in the GISAXS simulations. The existing codes can only treat simple collections of shapes whose form factors can be derived analytically.

We begin with a brief introduction to the theory behind form factor in DWBA. A detailed description can be found in [6]. The scattering intensity of the X-rays obtained is represented as

$$I(\vec{\mathbf{q}}) = \frac{k_0^4}{16\pi^2}|\Delta n^2|^2|\Phi(\vec{\mathbf{q}}_{||}, k_{zi}^0, k_{zf}^0)|^2, \tag{1}$$

where $\Delta n^2$ is the difference in the refractive indices of the particle and the substrate, and for a nanoparticle supported over the substrate surface,

$$\begin{aligned}\Phi(\vec{\mathbf{q}}_{||}, k_{zi}^0, k_{zf}^0) &= F(\vec{\mathbf{q}}_{||}, k_{zf}^0 - k_{zi}^0) \\ &+ r_{0,1}^f F(\vec{\mathbf{q}}_{||}, -k_{zf}^0 - k_{zi}^0) \\ &+ r_{0,1}^i F(\vec{\mathbf{q}}_{||}, k_{zf}^0 + k_{zi}^0) \\ &+ r_{0,1}^i r_{0,1}^f F(\vec{\mathbf{q}}_{||}, -k_{zf}^0 + k_{zi}^0).\end{aligned} \tag{2}$$

Here, $F$ is the form factor, and the four terms represent the different refelction-refraction cases. The form factor of $\vec{\mathbf{q}}$ is given by

$$F(\vec{\mathbf{q}}) = \int_{S(\vec{r})} e^{i\vec{\mathbf{q}}\cdot\vec{r}} d\vec{r}, \tag{3}$$

where the integral is over the shape function of the nanoparticles in the sample. For computational purposes, the shape surface is triangulated and the form factor is then approximated as the summation over all the generated triangles. If $s_t$ is the surface area of a triangle $t$, the form factor can be written as

$$F(\vec{\mathbf{q}}) = \sum_{t=1}^{N} e^{i\vec{\mathbf{q}}\cdot\vec{r}} s_t \tag{4}$$

where $N$ is the total number of triangles. In Figure 2, two example form factor intensity images are shown for simple shapes, a cylinder and a sphere. Because of the simplicity of these structures, the images have been analytically computed.

We have implemented an efficient and flexible GISAXS simulation code based on the DWBA theory with C++, Nvidia's CUDA, and MPI on a cluster of GPUs. Our code can compute the scattered light intensity from any given sample in all directions of space; thus allowing full construction of the GISAXS pattern. The software allows simulating diffraction image for any given superposition of custom shapes or morphologies (e.g. obtained graphically via a discretization scheme) in a user-defined region of $k$-space (or region of the area

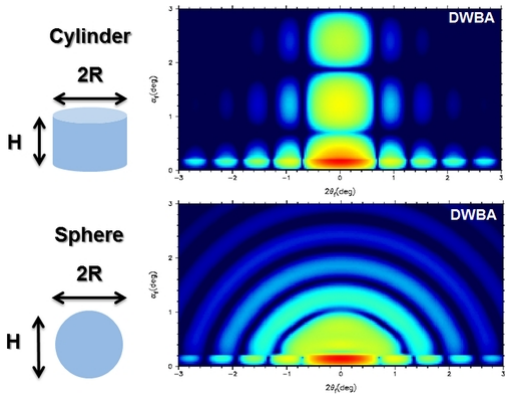


Figure 2: Simulated form factors for a cylinder ($R = H$ =5nm), and a sphere ($R$ =5nm, $H$ =10nm.) *Images taken from A. Meyer's www.gisaxs.de*

detector) for all possible grazing incidence angles and in-plane sample rotations. This flexibility allows to easily tackle a wide range of possible sample geometries such as nanostructures on top of or embedded in a substrate, or a multilayered structure. In the following we describe our algorithm for computing the form factors on graphics processors.

## 2.1 Form Factor Kernel on GPUs

Calculation of the form factor at a point involves integral over the nanoparticle shape, approximated as a summation over the triangulated structure (Equation 4). The number of triangles also corresponds to the complexity and resolution of the nanostructure under consideration. Given a user-defined region in $k$-space, $Q$-grid, this form factor needs to be computed for each point in the grid. Computationally this problem can be defined as follows: Given a user-defined 3-dimensional $Q$-grid, of resolution $n_x \times n_y \times n_z$, and a set of $N$ triangles representing a triangulated nanostructure, we want to compute $F(\vec{\mathbf{q}})$ for each $q$-point $\vec{\mathbf{q}}$ in the $Q$-grid, thereby constructing $M$, a 3-D matrix of dimensions $n_x \times n_y \times n_z$.

Typically, $n_x$ is in the order of few hundreds, $n_y$ and $n_z$ in hundreds to thousands, and $N$ may range from few hundreds to millions. The computations of $F(\vec{\mathbf{q}})$ for all $q$-points are independent of each other, and there may be a large number of such points, making this application an ideal candidate for parallelization on graphics processors by efficiently utilizing the fine-grained parallelism offered by them. The computation of a form factor is divided into two phases, where for each $q$-point $\vec{\mathbf{q}}$,

1. first we compute the inner term, $F_t(\vec{\mathbf{q}}) = e^{i\vec{q}\cdot\vec{r}} s_t$ in Equation 4 for each triangle $t$, generating an intermediate array of size $N$,

2. followed by a reduction of this intermediate array over all the triangles to

result in the final form factor, $F(\vec{\mathbf{q}}) = \sum_t F_t(\vec{\mathbf{q}})$.

Apart from being compute-intensive, these computations are memory-demanding as well. Firstly, the size of the matrix $M$ is generally large, where the number of $q$-points can range from a million to hundreds of millions and the number of triangles can range from a few hundreds to millions. In addition, the first phase of the computations generates an intermediate 4-dimensional matrix $M_I$, where for each $q$-point $(q_x, q_y, q_z)$, the fourth dimension corresponds to the set of input triangles $\{t_0, \cdots, t_{N-1}\}$ as mentioned above, thereby further increasing memory usage by a factor of $N$. Therefore, careful memory management is crucial for handling such computations.

### 2.1.1 Basic Implementation

In the first phase, computation of the inner term for each triangle at a $q$-point is independent of the computations for all other triangles at the same $q$-point. A basic implementation of the first phase may exploit this data parallelism to parallelize the computations across the triangles. To do so we define a CUDA thread block here as a one-dimensional array of threads, $T_s, \cdots, T_e$. Let the size of a thread block be $B_t$, then the number of thread blocks hence generated would be $\lceil \frac{N}{B_t} \rceil$. Each thread from all thread blocks is mapped to a unique input triangle. A single CUDA thread $T_i$ is hence responsible for a particular triangle $t_j$ across all $q$-points. The mapping can be defined as

$$T_i \xrightarrow{map} t_j, 0 \leq j \leq N-1. \tag{5}$$

One mapping can simply be $T_i \xrightarrow{map} t_i$ by defining $i = j$ for all $j$. $T_i$ computes the inner value $F_{t_j}(\vec{\mathbf{q}})$ for each of the $n_x n_y n_z$ points in the $Q$-grid. An illustration of this decomposition and mapping is shown in Figure 3.

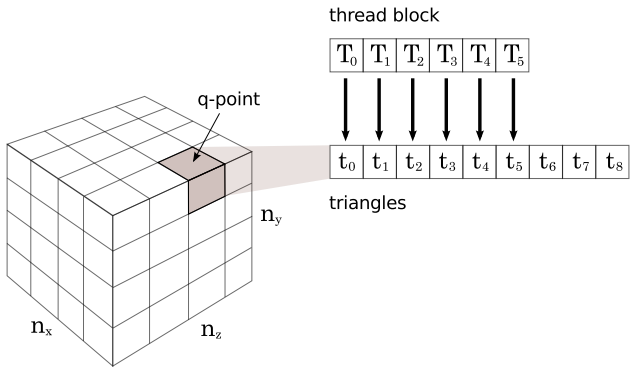


Figure 3: Phase 1 – Decomposition of computations during the first phase is done along the triangles. Each CUDA thread is mapped to a unique triangle. A triangle is a coordinate in the fourth dimension for all $q$-points in the $Q$-grid.

The second phase involves sum-reduction of the intermediate values computed in first phase. For each $q$-point $\vec{\mathbf{q}} = (q_x, q_y, q_z)$, the intermediate values $F_{t_j}(\vec{\mathbf{q}})$ are summed over all the triangles $t_j$ $(0 \leq j \leq N)$ corresponding to this $q$-point, to result in the form factor $F(\vec{\mathbf{q}})$.

Since in this phase the reduction is over the fourth dimension (triangles), we can no longer implement the parallelism along this dimension as in the first phase. Therefore, in this phase we exploit the independence of each $q$-point and parallelize the computations along the three $x$, $y$, and $z$ dimensions. The computation of $M$ is decomposed into a grid consisting of equally sized 3-dimensional blocks. On the GPU, these blocks correspond to CUDA thread blocks, with each thread $T_{i,j,k}$ mapped to a unique $q$-point $(q_{x_i}, q_{y_j}, q_{z_k})$. A simple mapping in this case can be

$$T_{i,j,k} \xrightarrow{map} \vec{\mathbf{q}}_{i,j,k} = (q_{x_i}, q_{y_j}, q_{z_k}). \tag{6}$$

An example of this decomposition and mapping is shown in Figure 4. Thread $T_{i,j,k}$ is responsible to compute the final form factor value $F(\vec{\mathbf{q}}_{i,j,k})$ by summing $F_{t_l}(\vec{\mathbf{q}}_{i,j,k})$ over each input triangle $t_l$, $0 \leq l \leq N-1$.

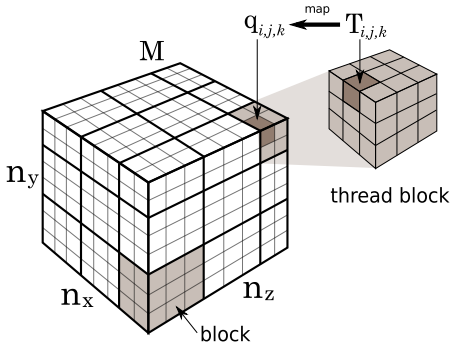


Figure 4: Phase 2 – Decomposition of $M$ into *blocks*, and mapping of CUDA threads to the $q$-points. Each thread is responsible for the reduction over all the triangles at its mapped $q$-point.

### 2.1.2 Handling Memory Limitations

Due to large memory requirements during these computations even for moderately sized $Q$-grids (as mentioned above), as well as limited sizes of the device and host memories, a careful use of memory space is an essential key to obtaining high-performance on the GPUs for form factor computations. We address this issue of handling large $Q$-grid sizes and number of triangles by once more taking advantage of the data parallelism present in the form factor computations at each $q$-point.

We decompose the intermediate 4-D matrix $M_I$ along each of the four dimensions into a number of equally sized (except in boundary cases) disjoint

four-dimensional *hyperblocks*, covering all the $q$-points and triangles. Let the size of a hyperblock, $M_h$, be represented by $h_x \times h_y \times h_z \times h_t$, where $h_x$, $h_y$, $h_z$ and $h_t$ are the side-lengths of $M_h$ in the $x$, $y$, $z$ and $t$ dimensions respectively ($h_\alpha \leq n_\alpha$, $\alpha \in \{x, y, z, t\}$). The maximal set of hyperblocks, where each hyperblock contains the same $q$-points (but different triangles,) can be uniquely mapped to the 3-dimensional matrix $M$: All the hyperblocks in this maximal set from $M_I$ map to a single *block*, $M_b$, of size $b_x \times b_y \times b_z$ in $M$ where $b_x = h_x$, $b_y = h_y$, $b_z = h_z$, and the coordinates of the $q$-points in this block are equal to those of the hyperblocks. This is illustrated in Figure 5. The total number of these hyperblocks constructed in $M_I$ is equal to $\left\lceil \frac{n_x}{b_x} \right\rceil \left\lceil \frac{n_y}{b_y} \right\rceil \left\lceil \frac{n_z}{b_z} \right\rceil \left\lceil \frac{N}{b_t} \right\rceil$, and the number of corresponding blocks in $M$ would be $\left\lceil \frac{n_x}{b_x} \right\rceil \left\lceil \frac{n_y}{b_y} \right\rceil \left\lceil \frac{n_z}{b_z} \right\rceil$.

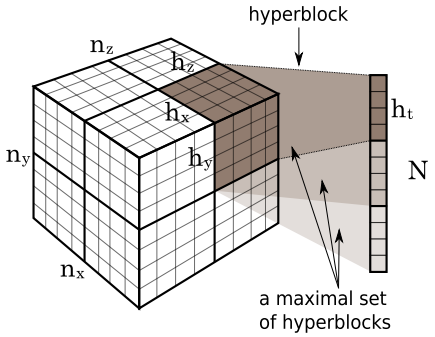


Figure 5: Decomposition of $M_I$ into *hyperblocks*. The maximal sets of such hyperblocks corresponding to the same set of $q$-points, but different triangles, are mapped to a unique *block* in the matrix $M$.

The idea here is to decompose the computations such that a resulting hyperblock can be completely handled in the available device memory. At the minimum we need to store the intermediate 4-D matrix $M_I$ and the final form factor matrix $M$. (We can ignore the other input sets since their sizes are small compared to the size of $M$.) Therefore, the memory usage is at the least $cn_xn_yn_z(N+1)$ bytes, where $c$ is a constant representing the number of bytes used to encode a single value. Now suppose we decompose the intermediate matrix $M_I$ into hyperblocks as above, then the memory requirement to process one hyperblock would be $ch_xh_yh_z(h_t+1)$ bytes. Hence, smaller the size of a hyperblock, the lesser memory it would require. Note that we can easily decompose the computations along the fourth dimension $t$ by dividing the reduction phase into two steps using the fact that the summation operation is both associative

and commutative:

$$F(\vec{\mathbf{q}}) = \sum_{t=0}^{N-1} F_t(\vec{\mathbf{q}}) \quad (7)$$

$$= \sum_{u=0}^{\lceil \frac{N}{b_t} \rceil - 1} \left( \sum_{t=0}^{b_t - 1} F_t(\vec{\mathbf{q}}) \right). \quad (8)$$

Partial reductions along the dimension $t$ are computed for a hyperblock resulting in a 3-dimensional matrix $M_p$. The number of such partially reduced hyperblocks $M_p$ corresponding to a $M_b$ is equal to $\left\lceil \frac{N}{b_t} \right\rceil$. The maximal set of partial matrices $M_p$, which map to the same block $M_b$, are then reduced to construct the final output block $M_b$ in matrix $M$. In other words, we can view this phase of computations as first reducing the size of the fourth dimension from $N$ to $\left\lceil \frac{N}{b_t} \right\rceil$, and then reducing the smaller sized fourth dimension to obtain a 3-dimensional matrix $M_b$.

### 2.1.3 Algorithm Overview

The overall scheme for computing form factors of each $q$-point in the input $Q$-grid on a single CPU-GPU node can be described as the following steps.

**Initialization**

- Input $Q$-grid with resolution $n_x \times n_y \times n_z$.
- Input set of $N$ triangles: $\{t_0, \cdots, t_{N-1}\}$.
- Initialize output matrix $M$ of size $n_x \times n_y \times n_z$. Each output value in $M$ will be the form factor at the corresponding $q$-point.

**Computations**

1. Copy $Q$-grid resolutions and set of triangles to GPU device memory.
2. Calculate hyperblock size, $b_x \times b_y \times b_z \times h_t$.
3. Calculate number of hyperblocks = $\left\lceil \frac{n_x}{b_x} \right\rceil \left\lceil \frac{n_y}{b_y} \right\rceil \left\lceil \frac{n_z}{b_z} \right\rceil \left\lceil \frac{N}{b_t} \right\rceil$
4. For each hyperblock $M_h$:
   (a) Initialize $M_h$ in device memory.
   (b) Launch **Phase 1** kernel on GPU to compute $M_h$. A CUDA thread $T_i$ executes:
      i. Compute triangle index $j$ using $T_i \xrightarrow{map} t_j$.
      ii. For each $q$-point $\vec{\mathbf{q}} = (q_x, q_y, q_z)$:
         - Compute $F_{t_j}(\vec{\mathbf{q}}) = e^{i\vec{q}\cdot\vec{r}} s_{t_j}$.

- $M_h(q_x, q_y, q_z, t_j) = F_{t_j}(\vec{\mathbf{q}})$.

(c) Initialize $M_p$ of size $b_x \times b_y \times b_z$ in device memory.

(d) Launch **Phase 2** (reduction) kernel on GPU to compute $M_p$. A CUDA thread $T_{i,j,k}$ executes:

i. Compute $q$-point coordinates $(q_x, q_y, q_z)$ using $T_{i,j,k} \xrightarrow{map} \vec{\mathbf{q}}_{i,j,k}$.

ii. Compute $M_p(q_x, q_y, q_z) = \sum_{l=0}^{h_t} M_h(q_x, q_y, q_z, t_l)$

(e) CPU copies $M_p$ from device memory and adds each value to the values at the corresponding indices in $M$. At the end, each such block will correspond to the matrix $M_b$.

5. Return output $M$, 3-dimensional array with each value equal to the form factor of the corresponding $q$-point.

#### 2.1.4 Choosing a Hyperblock Size

Till now we have assumed that we are already given the hyperblock size. We will not remove this assumption. One would expect to have the hyperblock size such that it fills the device memory as much as possible, since this would mean less number of hyperblocks, and hence smaller number of iterations in the algorithm. Partially reduced blocks $M_p$ are transferred from the device memory to the host memory. Since the data transfer bandwidth between host memory and the device memory is quite low ( 8 GB/s), even with overlapped data transfer and computations, this step becomes a bottleneck, thereby reducing performance.

On the other hand, keeping the hyperblock size too low also results in a degraded performance. A major factor for this is that the reduction phase derives its parallelism from the number of $q$-points in the hyperblock. Reducing its size would mean reducing the available parallelism, resulting in under utilization of the multiprocessors. Similarly, the first phase derives its parallelism from number of triangles $h_t$ in a hyperblock leading to the same effect.

As it turns out, the choice of the hyperblock size plays a crucial role in the performance of the code, affecting the runtimes by almost an order of magnitude. This size should neither be too big, nor too small. In order to demonstrate this, as well as to choose an optimal hyperblock size, we conducted extensive experiments by varying the four parameters $b_x$, $b_y$, $b_z$ and $h_t$. In the following we show some of these results. We use two datasets for these experiments: dataset A with 2,292 triangles, and dataset B with 91,753 triangles. We use a $Q$-grid of resolution $90 \times 200 \times 200$ (3.6M $q$-points).

Since the first phase depends on the number of triangles alone for parallelism, in these experiments shown we keep $h_t$ constant at 2,000, and vary only $b_y$ and $b_z$. $n_x$ is typically small compared to $n_y$ and $n_z$, hence we assign $b_x = n_x = 90$ in this case. In Figure 6, we show a heat-map for dataset A and in Figure 7 for dataset B. The warmer/lighter the color, the more execution time is taken for the computations. The cooler/darker the color, the faster the computations were done. All the execution times shown are in seconds. We note that we get

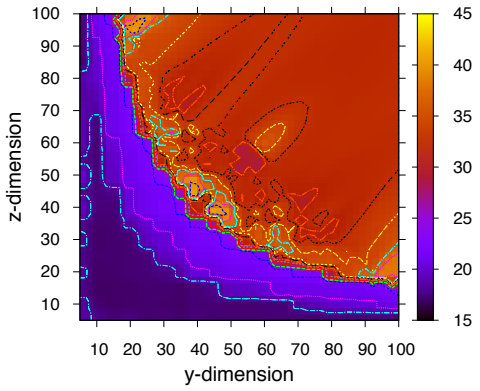


Figure 6: Heat map of execution times with varying hyperblock sizes for the dataset with $N = 2,292$. On the $x$-axis is the hyperblock size along the $y$-dimension, and on $y$-axis is the size along $z$-dimension. The darker/bluer regions are where optimal performance is achieved.

optimal performances towards the lower sizes of $b_y$ and $b_z$, but keeping them too low again increases the runtimes, as can be seen on the lower left corners of the two graphs. Based on extensive similar experiments (also with variable $b_x$ and $h_t$), we selected the hyperblock size parameters $b_x = n_x$, $b_y = 20$, $b_z = 15$, and $b_t = 2,000$. We use these parameter values for conducting further experiments and performance analyses.

### 2.1.5 Performance on a GPU

In this subsection we will present some of the runtime results on a GPU. We used a system with Nvidia Tesla M2090 graphics processor [4]. This graphics card has 5 GB device memory, 512 CUDA cores across 16 multiprocessors (32 each), and 48 KB shared memory per block. The clock speed is 1.3 GHz. This GPU is attached to a dual-socket 2.93 GHz Intel Xeon X5670 processor with a total of 12 cores, and 90 GB main memory. We implemented the described algorithm in C++ with Nvidia CUDA.

For the results shown here, we use three datasets: datasets A and B described in the previous subsection, with 2,292 and 91,753 triangles respectively; and dataset C with 6,600 triangles. We use $Q$-grids of two resolutions: $90 \times 200 \times 200$ (3.6M $q$-points), and $90 \times 800 \times 800$ (57.6M $q$-points). In order to assess the performance of our scheme, we also implemented a cache-optimized sequential version on CPU. This was also run on the same system. Table 1 contains the execution times in seconds for the three datasets and 3.6M $q$-points, on a GPU and the sequential runtime on a CPU. Table 2 contains the runtimes

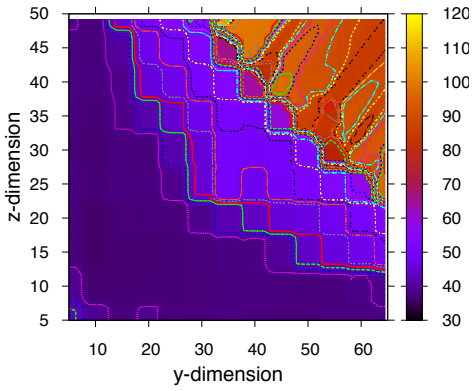


Figure 7: Heat map of execution times with varying hyperblock sizes for the dataset with $N = 91,753$. On the $x$-axis is the hyperblock size along the $y$-dimension, and on $y$-axis is the size along $z$-dimension. The darker/bluer regions are where optimal performance is achieved.

Table 1: Execution times in seconds for the three datasets, and $Q$-grid of resolution $90 \times 200 \times 200$ (3.6M $q$-points).

| # triangles | GPU runtime | Sequential | Speedup |
|---|---|---|---|
| 2,292 | 14.76 | 996.62 | 67.52 |
| 6,600 | 30.85 | 2895.76 | 93.86 |
| 91,753 | 370.06 | 40994.64 | 110.78 |

in seconds for the three datasets and 57.5M $q$-points on a GPU. Sequential runtimes and corresponding speedups are also given. The sequential code was unable to execute with the higher resolution $Q$-grid on the third dataset (91,753 triangles).

### 2.1.6 Utilizing a Cluster of GPUs

With the amount of data generated (e.g. 100 megabytes per second as mentioned in Section 1,) there is an urgent need to be able to analyze this data in real-time. Clearly our implementation on a single GPU node cannot match this need. Further, to have a higher resolution structure prediction of nanoparticles, a higher resolution $Q$-grid is needed. Increasing the number of $q$-points in each of the three dimensions increases the complexity and memory usage by $O(n^3)$. Hence, we need capability to handle much larger sizes of the form factor array $M$, which may not fit into the available system memory on a single node system.

Given the aforementioned requirements to analyze data faster, and handle

Table 2: Execution times in seconds for the three datasets, and $Q$-grid of resolution $90 \times 800 \times 800$ (57.6M $q$-points).

| # triangles | GPU runtime | Sequential | Speedup |
|---|---|---|---|
| 2,292 | 234.09 | 21205.68 | 90.59 |
| 6,600 | 492.95 | 61998.36 | 125.77 |
| 91,753 | 5865.67 | NA | NA |

higher resolutions, we utilize distributed-memory supercomputers/clusters with GPU accelerators at each node, to move a step closer to these goals. This requires adding another level of parallelism to our scheme which would decompose computations across the nodes in the cluster. We use the message-passing model to achieve parallelism across multiple nodes.

In a typical scenario $n_x$ is usually small, hence the resolution is mostly determined by $n_y$ and $n_z$. We use this knowledge to decompose the $Q$-grid along the two dimensions $y$ and $z$. Suppose we have $p$ nodes available. We divide the to be computed $M$, corresponding to the $Q$-grid, into a 2-dimensional grid of equally-sized submatrices. The size of this grid is $\lfloor\sqrt{p}\rfloor \times \frac{p}{\lfloor\sqrt{p}\rfloor}$ along the $y$ and $z$ dimensions respectively. Hence, when $p = q^2$, the grid is $q \times q$ sized. Let us call a resulting division of the $Q$-grid a *$Q$-tile*, and corresponding submatrix of $M$ simply a *tile*. Size of a $Q$-tile is $n_x \times {}_pn_y \times {}_pn_z$ where

$$ {}_pn_y = \frac{n_y}{\lfloor\sqrt{p}\rfloor}, {}_pn_z = \frac{n_z}{\frac{p}{\lfloor\sqrt{p}\rfloor}}. $$

Each of the nodes $P_{i,j}$ is assigned to compute a distinct tile $M_{k,l}$ through a mapping

$$ P_{i,j} \xrightarrow{map} M_{k,l}, 0 \leq i \leq \lfloor\sqrt{p}\rfloor - 1, 0 \leq j \leq \frac{p}{\lfloor\sqrt{p}\rfloor} - 1. \tag{9} $$

In a simple mapping, we set $k = i$ and $l = j$. Figure 8 illustrates this decomposition of the matrix $M$ into tiles.

At the initialization, $P_{i,j}$ reads its assigned $Q$-tile corresponding to $M_{i,j}$. With the above scheme, since the form factor computations are independent along the $y$ and $z$ dimensions, the problem is now decomposed into independent sub-problems for each node in the cluster to compute. Each node $P_{i,j}$ proceeds to use our single node algorithm from the previous subsection to compute $M_{i,j}$. Once completed, an assigned master node may gather computed tiles from other processors and put them together to form the final form factor array $M$. In the following we give an overall overview of our algorithm executed by each node $P_{i,j}$:

**Initialization at $P_{i,j}$**

- Calculate values ${}_pn_y$ and ${}_pn_z$ and Input a unique and disjoint $Q$-tile $(i, j)$ of resolution $n_x \times {}_pn_y \times {}_pn_z$.

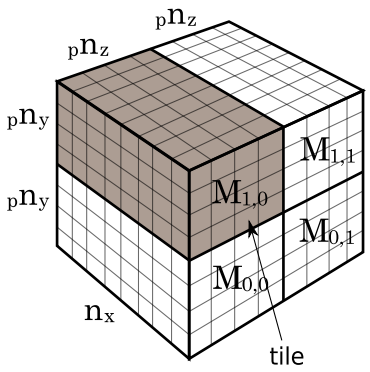


Figure 8: Decomposition of $Q$-grid and $M$ into *tiles*. Each $M_{i,j}$ is assigned to processor $P_{i,j}$ for computations. In this example, $p = 4$.

- Input set of $N$ triangles: $\{t_0, \cdots, t_{N-1}\}$.
- Initialize output matrix $M_{i,j}$ of size $n_x \times {}_pn_y \times {}_pn_z$.

**Computations at** $P_{i,j}$

1. Use the algorithm from Section 2.1.3 with the input $Q$-tile $(i, j)$ as the local $Q$-grid of resolution $n_x \times {}_pn_y \times {}_pn_z$ to perform computations using local GPU.
2. Construct local output array $M$.
3. Return the computed local $M$ as $M_{i,j}$ to $P_{0,0}$.

**Finalization at** $P_{0,0}$

1. Gather $M_{i,j}$ from all $P_{i,j}$.
2. Place each tile at its correct place forming $M$ and write this output.

When size of $M$ is too large, a single node gathering outputs from all other processors is infeasible. Each processor may then directly write its output at correct position in the common storage/disk.

### 2.1.7 Performance on a GPU Cluster

We implemented our algorithm using C++ with MPI for the topmost level of parallelism which distributes the computational work across the nodes in the cluster. Here we show performance results of our algorithm on a GPU cluster. This GPU cluster consists of 42 available nodes connected with InfiniBand interconnects. Each node consists of a dual-socket Intel 5530 Nehalem processors with 2.4 GHz clock with a total of 8 cores per node. This has 24 GB main memory. Each node has an Nvidia Tesla C2050 (Fermi) GPU with 3 GB device

Table 3: Execution times in seconds for the three datasets, and $Q$-grid of resolution $90 \times 200 \times 200$ (3.6M $q$-points) on varying number of GPU nodes.

| # GPU Nodes | $N = 2,292$ | $N = 6,600$ | $N = 91,753$ |
|---|---|---|---|
| 1 | 17.01 | 35.03 | 428.88 |
| 2 | 8.62 | 17.67 | 215.35 |
| 4 | 4.37 | 8.95 | 108.35 |
| 8 | 2.27 | 4.65 | 55.01 |
| 12 | 1.58 | 3.13 | 36.44 |
| 16 | 1.23 | 2.43 | 27.70 |
| 24 | 1.00 | 1.69 | 18.62 |
| 30 | 0.83 | 1.45 | 15.20 |
| 36 | 0.73 | 1.28 | 12.64 |
| 42 | 0.66 | 1.12 | 10.83 |

Table 4: Execution times in seconds for the three datasets, and $Q$-grid of resolution $90 \times 800 \times 800$ (57.6M $q$-points) on varying number of GPU nodes.

| # GPU Nodes | $N = 2,292$ | $N = 6,600$ | $N = 91,753$ |
|---|---|---|---|
| 1 | 268.90 | 559.73 | 6794.21 |
| 2 | 134.92 | 280.63 | 3401.18 |
| 4 | 67.88 | 140.90 | 1705.44 |
| 8 | 34.40 | 71.57 | 856.48 |
| 12 | 23.12 | 47.89 | 569.73 |
| 16 | 17.54 | 36.27 | 429.33 |
| 24 | 11.94 | 24.38 | 286.42 |
| 30 | 9.73 | 19.65 | 229.71 |
| 36 | 8.23 | 16.50 | 191.60 |
| 42 | 7.21 | 14.37 | 164.51 |

memory. One such GPU has a total of 448 CUDA cores across 14 multiprocessors (32 each). Each block has 64 KB of L1 cache and shared memory.

In the following we present some of the results of our experiments on this GPU cluster. We use the same datasets A, B and C as previously, and the two resolutions of the $Q$-grid. Table 3 shows execution times in seconds, for computing the form factor of the three datasets on $Q$-grid of $90 \times 200 \times 200$ resolution, with varying number of nodes. Strong scaling data for these results are shown in Figure 9 as a speedup graph. Table 4 shows execution times in seconds for a $Q$-grid of $90 \times 800 \times 800$ resolution with varying number of nodes. The speedup graph for this result is shown in Figure 10.

The larger the number of triangles and/or $q$-points are, the greater is the degree of parallelism available during the computations. Phase 1 of the algorithm on the GPU achieves fine-grained parallelism from the number of triangles, while phase 2 depends on the number of $q$-points. We see this in the results where the speedup obtained improves as the dataset size and/or $Q$-grid resolution is

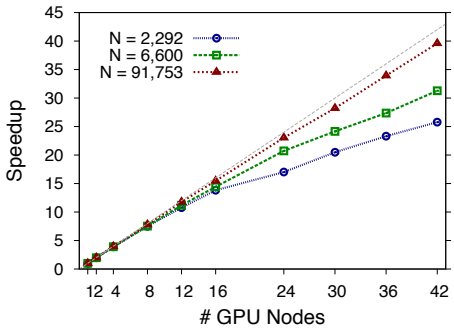


Figure 9: Relative speedups with varying number of GPU nodes, for the three datasets, with $Q$-grid of size $90 \times 200 \times 200$.

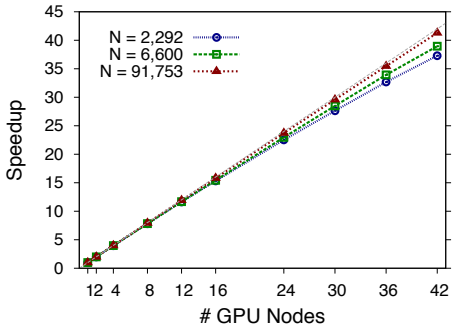


Figure 10: Relative speedups with varying number of GPU nodes, for the three datasets, with $Q$-grid of size $90 \times 800 \times 800$.

increased. We achieve a speedup of 41.3 on 42 GPU nodes for the dataset with 91,753 triangles, and 57.6M $q$-points.

This parallel multi-GPU code is capable of computing GISAXS images from much larger samples and with much higher resolutions than what were previously possible using sequential codes.

# 3 Reverse Montecarlo Modelling

Reverse Monte Carlo (RMC) modeling is a general method of structural modeling based on experimental data [3, 2]. It has become one popular analysis method used to extract information on material structure from small angle X-ray scattering (SAXS) data. RMC is a variation of standard Monte Carlo method employed in inverse modeling. In inverse modeling, the starting point is a set of experimental data, and the structural models (i.e., the atomic configurations) are generated by a procedure explicitly designed to give best matching

with experimental data in statistical sense. In this fitting procedure, one attempts to simulate various configurations of the underlying atoms, molecules, or building blocks (e.g., nanoparticles) in a material until the scattering pattern from the simulated structure matches the real scattering data, such as the experimentally measured structure factor.

In the X-ray scattering case, for each sample being probed, hundreds of 2D scattering patterns are generated with different angles of the beams and the rotations of the sample. In RMC modeling, we start with an initial configuration of the particles. In our case, this is a two-dimensional array of $N$ points (particle positions) in a square of side $L$. The array may be generated at random. We perform simultaneous simulations with multiple frames of image. In each simulation cycle, we first calculate the radial distribution function which represents the real-space structure. We then perform Fourier transform of the radial function which gives the reciprocal-space structure. The Fourier transform of the total scattering measurement provides information about the relative positions of the atoms. Then, we compare the computed structure factor with the real measured scattering data using a standard $\chi^2$ test. If the difference is large, we generate a new configuration by randomly moving one particle (point) to a valid position. Once the new configuration is generated, the simulation cycle is repeated until $\chi^2$ decreases to an equilibrium value and oscillates about it, as with the energy in a conventional Monte Carlo simulation. The resulting configuration should be a 3D structure that is consistent with the data within experimental errors. Figure 11 illustrates this simulation process.

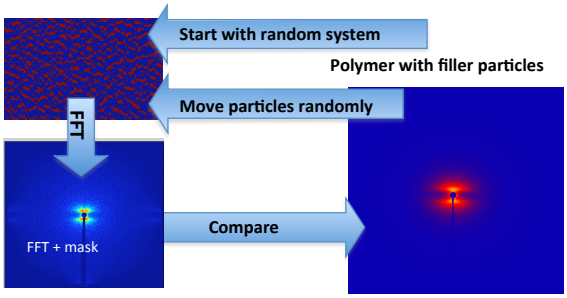


Figure 11: The Reverse Monte Carlo modeling.

The RMC procedure described above is very general and can be used to fit different quantities in many applications. In our case, we run a set of simultaneous RMC simulations to probe a parameter set corresponding to the loading of the lattice array. The following pseudocode shows our RMC simulation of one image frame. Each 2D image is represented as an $N \times N$ array. The typical size is 512 and the number of tiles is usually 100.

**Initialization**

- Input an image, find nonzero patterns of the images $P(:,:)$, which is the

measured quantity to be fitted with.

- Set the model loading factor for each lattice array, which is the fraction of the positions in the lattice array occupied by the particles.
- Create lattice with randomly filled positions initialized to 1, and empty positions to 0: $A(:,:) = 0$, $A(filled) = 1$. That is, $A$ is a 0/1 matrix.
- Calculate initial structure factor and $\chi^2$ error:
  $F_0(:,:) = \text{2D\_FFT}(A(:,:))$, $F_0^2 = cwise\_sqr(F)$,
  $\chi_0^2 = \sum_{i,j}(P(i,j) - F^2(i,j))$.

**Simulation steps:** $n = 1, ...,$ **until** $\chi_n^2$ **reaches equilibrium.**

1. Move one particle at random from old position $(i_{old}, j_{old})$ to new position $(i_{new}, j_{new})$.
2. Calculate the update from Fourier transform and new structure factor:
   $U = dft2(i_{old}, j_{old}, i_{new}, j_{new})$,
   $F_n = F_{n-1} + U, \quad F_n^2 = cwise\_sqr(F_n)$.
3. Mask the zero positions in $F_n^2$ and compute new $\chi^2$ error:
   $\chi_n^2 = \sum_{i,j}(P(i,j) - F_n^2(i,j))$.
4. If $\chi_n^2 < \chi_0^2$, the move is accepted, resulting in a new configuration. If $\chi_n^2 > \chi_0^2$, the move is accepted with a probability that follows a normal distribution. Otherwise, the move is rejected.
5. Repeat from step 1, either with the new configuration or the old one.

In this procedure, $cwise\_sqr(F)$ is a function that squares each element of array $F$ component-wise. The purpose of $dft2()$ function is to compute the update to the old structure factor. Note that when we start the simulation, it is necessary to calculate $F$ by a full Fourier transform (c.f., $2D\_FFT()$ function at initialization). However, between the two successive simulation steps, since only one particle is moved, the change to $F$ is rather small, it is only necessary to calculate the change in $F$ instead of a full FFT operation. We now explain how this is done in $dft2()$ procedure. Recall that the discrete Fourier matrix of size $N$ is $D = (d_{m,n})_{N\times N}$, where an $(m,n)$ entry $d_{m,n}$ is defined as: $d_{m,n} = \omega_N^{m\cdot n}$, where $\omega_N = e^{-2\pi i/N}$ is a primitive $N$th root of unity. The standard 2D Fourier transform for the structure factor $F(:,:) = 2D\_FFT(A(:,:))$ computes:

$$f_{k_1,k_2} = \sum_{n_1=0}^{N-1} \left( \omega_N^{k_1 n_1} \sum_{n_2=0}^{N-1} \omega_N^{k_2 n_1} a_{n_1,n_2} \right). \tag{10}$$

This requires $O(N^2 \log N)$ operations. However, since only one particle is moved in each step, the position matrix $A$ has only two entries changed:

$$a_{i_{old},j_{old}} = 1 \to 0, \quad a_{i_{new},j_{new}} = 0 \to 1.$$

Relating this to Eqn.(10), the change to matrix $F$ amounts to adding the new contribution due to $a_{i_{new},j_{new}} = 1$ and subtracting the old contribution due to $a_{i_{old},j_{old}} = 0$. These changes can be computed from two rank-1 matrices derived from two columns and rows of the DFT matrix $D$. One is the outer-product of $i_{old}$th column $D(:, i_{old})$ and $j_{old}$th row $D(:, j_{old})$, and the other is the outer-product of $i_{new}$th column $D(:, i_{new})$ and $j_{new}$th row $D(:, j_{new})$. Therefore, the function $dft2()$ computes the following update matrix, as needed in step 2 of the simulation cycle:

$$U = D(:, i_{new}) \cdot D(j_{new}, :) - D(:, i_{old}) \cdot D(j_{old}, :)$$

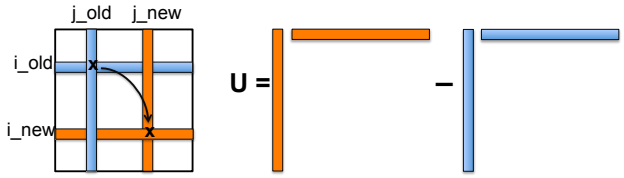


Figure 12: Moving a random particle (left), and the update matrix $U$ for the lattice Fourier matrix (right).

Figure 12 illustrates the particle movement and its effect on the update matrix $U$. Computing the two rank-1 matrices requires $O(N^2)$ operations, which is considerably cheaper than that of the full FFT.

The RMC simulation for our application typically involve $10^4$ DFT2 kernels to be computed per iteration step, and the total number of iterations can approach $10^6$. The initial in-house code was written in MATLAB and ran on one processor. In an earlier work, Scholz et al. ported this code to use Jacket to perform the simulations on GPU [7]. Jacket is a MATLAB wrapper developed by AccelerEyes (www.accelereyes.com), which accelerates the MATLAB code on GPUs with minimal knowledge and time. Jacket contains the language processing system, JIT compiler, and runtime system to enable access to GPU from MATLAB's M-codes. It automatically translates M-code to high performance primitives required for best utilization of GPUs. The runtime system launches GPU kernels and optimizes memory transfers. This wrapper is easy to use, and abstracts away all of the complexity of GPU and CUDA programming from the user. The programming model extends M-language with a new class **g** objects. The operations on **g** objects are translated into GPU-enabled MEX code. An example M-code in MATLAB/Jacket is given below:

```
N = 128; // matrix size
M = 200; // number of tiled matrices

// Create Data
Ac = complex( gones(N,N,M, 'single'),0 );
Bc = complex( gones(N,N,M, 'single'),0 );
```

```
// Compute 200 (128x128) FFTs
gfor i = 1:M
    Ac(:,:,i) = fft2(Bc(:,:,i));
gend
```

In this example, Ac and Bc are two complex matrices of ones. Then 200 FFTs are performed. The M-code syntax **gones** is similar to **ones**, i.e., generate matrices of all ones. But **gones** indicates that the matrix is an **g** object, hence Jacket will put Ac and Bc on the GPU device memory. Similarly, **gfor** means the **for**-loop occurs on the GPU device.

The Jacket-enabled MATLAB code has already given tremendous performance boost for many MATLAB codes. In particular, Scholz's GPU-enabled RMC code has achieved over **100x** speedup in computing time using one GPU over the pure MATLAB code on one CPU. Despite this speed benefit, MATLAB/Jacket code has some limitations: it cannot handle large data set, and the GPU utilization may not be fully realized from the Jacket wrapper (e.g., insufficient code optimization). Recently, we have ported the MATLAB code to C++ code, enhanced with NVIDIA's CUDA. We now describe our C++/CUDA implementation.

As we examine the simulation steps 1-4, it is clear that steps 1 and 4 have no parallelism and can be done quickly on the CPU host. Most of the computations occur in steps 2 and 3, which also have ample data parallelism and are amenable to GPU acceleration. Therefore, we designed several GPU kernel routines to compute those quantities on the device, including the computations of $dft2$ update matrix $U$, $F_n$, $F_n^2$ and $\chi_n^2$. We store in GPU memory the following four matrices: the image pattern $P$, the Fourier space lattice matrices $F$ and $F^2$ and the DFT matrix $D$. For all the kernels, we arrange both the grid blocks and the thread blocks to be one-dimensional. In particular, we choose 512 to be the number of threads in a thread block, and $N^2/512$ to be the number of grid blocks. Since each matrix has $N^2$ elements, such an 1D organization assigns one thread to compute one element of the matrix. For the most part, the threads can be executed independently. Only for $\chi^2$ computation, we need to perform a sum-reduction operation in the end; we use the THRUST library to accomplish this.[1]

With this CUDA-enhanced C++ code, we have observed additional **9x** speedup over the MATLAB/Jacket code for the entire simulation using one GPU. Our future work is to design a good data distribution scheme in order to use multiple GPUs.

[1] Thrust is an open-source CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). http://thrust.googlecode.com.

# 4 Conclusions and Future Work

We have designed and implemented two classes of parallel algorithms to help the beam-line scientists and users at the Advanced Light Source to achieve real-time analyses of the X-ray scattering data. Our new DWBA code for simulating the GISAXS patterns has achieved more than 125x speedups on one GPU card compared to the sequential CPU code. Further parallelization across multi-GPU using MPI led to an additional 40x speedup on a 42-nodes GPU cluster. We also developed a new GPU-accelerated inverse modeling code based on the Reverse Monte Carlo method. We have demonstrated over 9x speedup over the previously developed Jacket-based GPU code, which significantly reduced the fitting time for morphology prediction.

In addition to tremendous runtime reduction, our new codes utilize the memory more efficiently, which allows much larger samples with higher resolutions to be simulated than what were previously possible using the old sequential code.

Our future work includes applying autotuning tools for optimal hyperblock size selection which is essential for highest performance on GPUs, multi-GPU parallelization of the RMC algorithm, and porting to even larger GPU clusters. In addition to continued optimization of these algorithms and codes, we are also collaborating with the other scientists to integrate this back-end computing engine into an automatic workflow management system, including an GUI input interface and visualization tools. This will allow ALS to truly harness the high-performance computing power.

# Acknowledgements

The authors acknowledge Alexander Hexmer, Slim Chourou, and Elaine Chan for providing the input datasets used in the experiments in this paper, as well as for collaborating on this application.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

# References

[1] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience*, 21(18):2457–2477, 2009.

[2] R. McGreevy. Reverse Monte Carlo modeling. *J. Phys.: Condens. Matter*, 13:R877–R913, 2001.

[3] R. McGreevy and L. Pusztai. Reverse monte carlo simulation: A new technique for the determination of disordered structures. *Molecular Simulation*, 1:6:359–367, 1998.

[4] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[5] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 4.0*, 2011.

[6] G. Renaud, R. Lazzari, and F. Leroy. Probing surface and interface morphology with grazing incidence small angle x-ray scattering. *Surface Science Reports*, 64:255–380, 2009.

[7] A. Scholz and A. Hexemer. Reverse Monte Carlo in Matlab/Jacket. Private communication, 2011.