# Massively Parallel X-ray Scattering Simulations

Abhinav Sarje*, Xiaoye S. Li*, Slim Chourou*, Elaine R. Chan† and Alexander Hexemer†
*Computational Research Division †Advanced Light Source
Lawrence Berkeley National Laboratory, Berkeley, CA
Email: {asarje, xsli, stchourou, erchan, ahexemer}@lbl.gov

***Abstract*—Although present X-ray scattering techniques can provide tremendous information on the nano-structural properties of materials that are valuable in the design and fabrication of energy-relevant nano-devices, a primary challenge remains in the analyses of such data. In this paper we describe a high-performance, flexible, and scalable Grazing Incidence Small Angle X-ray Scattering simulation algorithm and codes that we have developed on multi-core/CPU and many-core/GPU clusters. We discuss in detail our implementation, optimization and performance on these platforms. Our results show speedups of ~125x on a Fermi-GPU and ~20x on a Cray-XE6 24-core node, compared to a sequential CPU code, with near linear scaling on multi-node clusters. To our knowledge, this is the first GISAXS simulation code that is flexible to compute scattered light intensities in all spatial directions allowing full reconstruction of GISAXS patterns for any complex structures and with high-resolutions while reducing simulation times from months to minutes.**

## I. Introduction

X-ray scattering methods are a valuable tool for measuring the structural properties of materials used in the design and fabrication of energy-relevant nano-devices, such as photovoltaic, energy storage, battery, fuel, and carbon capture and sequestration devices. They 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 enable 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. In this paper we address the computational challenges in GISAXS data analysis. We obtain data from one such X-ray science facility – the Advanced Light Source (ALS) located at the Lawrence Berkeley National Laboratory (LBNL). This is a third-generation synchrotron light source and one of the world's brightest sources of ultraviolet and soft X-ray beams. It is a U.S. national user facility funded by the Department of Energy, and is internationally recognized for its world-class measurement capabilities in X-ray science.

Fig. 1 illustrates the GISAXS scattering geometry. An incident X-ray wave vector $\mathbf{k_i}$ is directed 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 $2\theta_\mathbf{f}$ 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 $2\theta_\mathbf{f}$. The incident angle $\alpha_\mathbf{i}$ can be varied and the sample can be rotated by an angle $\omega$ around its surface normal, thus creating many 2D images with various intensity profiles. Analysis algorithms are used to analyze these images and predict the atomic structure of the underlying sample being probed.

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. 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 the Linac Coherent Light Source (LCLS) facility in Stanford 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 and the accumulation of large amounts of data poses a severe impediment in designing a sequential set of studies. Consequently, researchers are faced with an extremely inefficient utilization of the 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.

In this work, we are developing new high-performance computing algorithms, codes, and software tools, targeting state-of-the-art HPC systems, for the analysis of X-ray scattering data collected at such beamline facilities. The targeted parallel platforms are large-scale parallel multi- and many-core systems with possibly hybrid node architectures, including GPU accelerators. In this paper, we present our recent parallel implementation and results for one of the most important class of the analysis algorithms used in the X-ray scattering community: the Distorted Wave Born Approximation (DWBA) model for GISAXS data simulations. Our new parallel package is called *HipGISAXS* (High Performance GISAXS), and will be released to the public in a couple of months. The most time-consuming task in the GISAXS simulations is the form factor calculation, and efficient implementation and optimization of

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

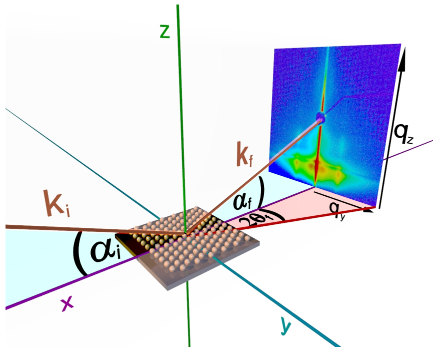

Fig. 1. Grazing incidence small angle X-ray scattering (GISAXS) geometry. *Graphic taken with permission from A. Meyer's www.gisaxs.de*

this kernel on above-mentioned targeted system architectures is the focal treatment of this paper.

## II. Related work

Presently the following three software packages are available for modeling GISAXS of coarse structures: *IsGISAXS* [1], *FitGISAXS* [2], and *DANSE* [3]. While these software suites incorporate the essential theoretical treatments of GI-scattering in the DWBA accurately for simple systems, they are severely limited for analyses and modeling of complex samples because of their rigorous input requirements for initial structural information. Hence, current GISAXS analysis is restricted to treatment of only a specific set of model shapes. Other factors, such as the platform dependencies of the packages and limitations on the levels of analysis available, have contributed to the lack of widespread use of these tools. For example, *IsGISAXS* presently runs only on the Microsoft Windows operating systems. Consequently, researchers have tended to abandon further investments in understanding the utilization of the software tools for their generic data analysis and modeling needs. Instead, they have resorted to writing their own analysis and simulation codes on case-by-case bases. These efforts require a considerable investment of time and resources, all the while increasing work duplications.

In addition to the above, an open source Python library entitled *PyNX* [4] has been recently released to help with a more precise computation of GISAXS patterns for disordered or distorted atomic structures. This library utilizes graphics processors (GPUs) to accelerate the computations of scattering events from structures with large numbers of atoms ($> 10^3$) in up to three dimensions, but a single simulation can run on a single GPU only, limiting the complexity and size of the inputs. *PyNX* has an advantage of allowing a user to upload custom atomic geometry as inputs for the simulations, but on the other hand, it can only treat structures which sit on top of the surface of a substrate and not those which are embedded within various media layers or buried within a substrate.

Our *HipGISAXS* codes provide a significant improvement over the other tools in several ways. *HipGISAXS* can compute the diffraction image for any given superposition of custom shapes or morphologies (for example, those obtained graphically via a discretization scheme), and for all possible grazing incidence angles and in-plane sample rotations. This flexibility permits the treatment of a wide range of possible custom structural geometries such as nanostructures. Furthermore, to our knowledge, *HipGISAXS* is the only GISAXS analysis and modeling code which can take advantage of state-of-the-art massively parallel hybrid many-core/GPU/CPU clusters and traditional multi-core/CPU clusters with tens of thousands nodes and hundreds of thousands cores, and is thus capable of reducing simulation times from months to minutes.

## III. The DWBA method

GISAXS is a unique method for investigating material topology and the structure of collections of nano-objects deposited on top of substrates or confined inside multilayered films. Simultaneous scanning of the in-plane and out-of-plane directions of the sample produce images that exhibit detailed features of the underlying nanostructures, hence allowing a wealth of information compared to alternative methods. To date the only theoretical framework modeling 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 [5].

One of the main objectives of GISAXS is to elucidate the features of highly complex nanostructures. This requires solving for *form factor* in a high-resolution $\mathbf{k}$-space grid, typically resulting in matrices with tens to hundreds of million grid-points. This time-consuming and memory-demanding calculation constitutes a major bottleneck in the GISAXS simulations. The existing codes described in Section II can only treat simple collections of shapes for which the form factors can be analytically computed.

We begin with a brief introduction to the theory behind the form factor in DWBA. A detailed description can be found in [5]. The scattering intensity of the X-rays obtained at a point $\vec{\mathbf{q}}$ in the $\mathbf{k}$-space is represented as

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

$\Delta n^2$ is the refractive index difference between the particle and the substrate; for a nanoparticle over a substrate surface,

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

where $F$ is the form factor, and the four terms represent the four different cases of refelction-refraction combinations. Form factor at a $q$-point $\vec{\mathbf{q}}$ is given by a surface integral as

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

The integral is over the shape surface of the nanoparticles in the sample under consideration. Computationally, the shape surface is discretized through triangulation, and the form factor is approximated as a summation over all the generated

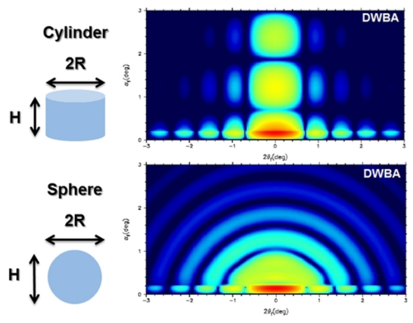

Fig. 2. Simulated form factors for a cylinder ($R = H = 5$nm), and a sphere ($R = 5$nm, $H = 10$nm.) *Graphics taken with permission from A. Meyer's www.gisaxs.de*

triangles. If $s_t$ represents the surface area of a triangle $t$, then the total 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. The higher the number of triangles (higher resolution), the better is the obtained approximation. In Fig. 2, two sample 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.

### A. Form Factor Kernel on HPC Systems

With the increasing rate of GISAXS data generation, as mentioned in Section I, there is an urgent need to be able to analyze the data sets in real-time because storing all the data is expensive and the amount of time required to carry out the analyses gets impractical. Furthermore, in future, the hardware will be incapable of transferring all the raw data collected at the detector due to the high data-generation rate. With this in mind, we have developed efficient and flexible GISAXS simulation codes based on the DWBA theory on high-performance systems as a step towards achieving the goal of real-time data analysis. In particular, we have developed codes on a hybrid cluster of GPUs with multi-core CPUs, and a cluster of purely multi-core CPUs. In the following sections, our implementation and analysis on these platforms will be discussed in detail. To our knowledge, this is the first GISAXS simulation code that is flexible enough to treat any custom complex morphologies, with high resolutions, all the while reducing the simulation times from months to minutes.

Recall that the main bottleneck kernel in the GISAXS simulation algorithm is the calculation of form factors, which involves integration over the nanoparticle shape, approximated as a summation over the discretized/triangulated shape surface (Equation 4). The number of triangles also corresponds to the complexity and resolution of the nanostructures under consideration. Given a user-defined region in the $k$-space as a $Q$-grid where the grid divisions may be irregular, the form factor needs to be computed for each point on this grid. Computationally our focal problem can be defined as follows:

*Given a user-defined 3-dimensional $Q$-grid of resolution $n_x \times n_y \times n_z$ grid-points, and a set of $N$ triangles representing the shape surface of 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 $n_x \times n_y \times n_z$.*

In a typical simulation, $n_x$ is on the order of a few hundreds, $n_y$ and $n_z$ on several hundred to thousands, and $N$ may range from a few hundred to millions. Note that the computation of $F(\vec{\mathbf{q}})$ for each $q$-point is independent of other $q$-points, making this application an ideal candidate for effective parallelization.

Apart from being compute-intensive, this problem is memory-demanding as well. First, the size of the matrix $M$ is generally large as mentioned above, with the number of $q$-points ranging from a million to hundreds of millions and the number of triangles ranging from few hundred to millions. This requires $O(n_x n_y n_z)$ memory to store the output. In addition, the computations generate an intermediate 4-dimensional matrix $M_I$, as will be described momentarily, where for each $q$-point $(q_x, q_y, q_z)$ the fourth dimension corresponds to the set of input triangles $\{t_0, \cdots, t_{N-1}\}$, thereby increasing memory usage by a factor of $N$. Also note that the computations are performed on complex numbers, doubling the memory requirements as opposed to real number computations.

To facilitate effective parallelization of this problem, we decompose the form factor computation into its primary components. Since the sum-reduction is over these components, we separate reduction from the main computational kernel. Specifically, we divide the computation of a form factor into two phases as follows. For a $q$-point $\vec{\mathbf{q}}$,

1) compute inner term $F_t(\vec{\mathbf{q}}) = e^{i\vec{q}\cdot\vec{r}} s_t$ (Eq. 4) for each triangle $t$, generating an intermediate array of size $N$,
2) sum-reduce the intermediate array over all the triangles, resulting in the final form factor, $F(\vec{\mathbf{q}}) = \sum_t F_t(\vec{\mathbf{q}})$.

Phase 1 generates an $N$ sized vector for each $q$-point, resulting in a 4-D matrix $M_I$ of size $n_x \times n_y \times n_z \times N$. Phase 2 performs sum-reduction over fourth dimension (triangles), generating the final form factor matrix $M$. We will now describe our parallelization strategies for these computations.

## IV. Parallelization on GPU clusters

In order to be parallelized, the computations need to be decomposed into subproblems. This is easy in our case due to the fact that there are no dependencies between the $q$-points for form factor computations. With a hierarchy of parallelism available in the system, our computations also need to be accordingly decomposed. As such, we begin in a top-down fashion where the first level of decomposition is across a cluster of GPUs. Computation on the $Q$-grid is distributed among all the computing nodes as follows.

### A. Across a GPU cluster

In a typical scenario $n_x$ is small – about one hundred or less. Hence the $Q$-grid resolution is mostly determined by $n_y$ and $n_z$ which, on the other hand, are typically large. We use this knowledge to decompose the $Q$-grid along the two dimensions $y$ and $z$, keeping $x$ intact. Suppose we have $p$ GPU nodes available. We divide the to be computed matrix $M$, into a two-dimensional grid of equally-sized sub-matrices. We take the

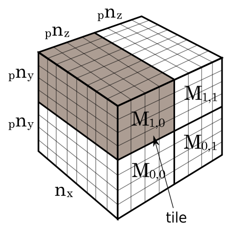

Fig. 3. Decomposition of $Q$-grid and $M$ into *tiles*. A tile $M_{i,j}$ is assigned to the processor $P_{i,j}$ for computations. In this illustration, $p = 4$.

size of this grid as $\lfloor\sqrt{p}\rfloor \times \frac{p}{\lfloor\sqrt{p}\rfloor}$ along the $y$ and $z$ dimensions respectively, and also arrange the compute nodes along the same way. 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 the corresponding sub-matrix of $M$ simply a *tile*. Let the size of a $Q$-tile be $n_x \times {}_pn_y \times {}_pn_z$ where

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

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

$$P_{i,j} \xrightarrow{map} M_{k,l}, 0 \leq i \leq \lfloor\sqrt{p}\rfloor - 1, 0 \leq j \leq \frac{p}{\lfloor\sqrt{p}\rfloor} - 1. \quad (5)$$

In a simple mapping we set $k = i$ and $l = j$. This scheme is illustrated in Fig. 3. At initialization, $P_{i,j}$ reads segments of the $q$-vectors, which define the $Q$-grid, corresponding to its assigned $Q$-tile. The problem is hence decomposed into independent sub-problems for each node in the cluster to compute. Each node $P_{i,j}$ proceeds to compute its tile $M_{i,j}$. Once completed, an assigned master node may gather computed tiles from other processors to form the final form factor matrix $M$. When $M$ is large, a single node gathering all outputs from other processors may become a bottleneck. To address this, each processor may directly write its output at its position in the common storage/disk through parallel I/O. Next we discuss how to perform the computations on each single GPU.

### B. On a Single GPU Node

Once a GPU is assigned a tile to compute, further decompositions are needed for parallelization on a GPU. In phase 1 of the computations, we utilize the fact that each computation of $F_t(\vec{\mathbf{q}})$ is independent of others along each dimension. Again, since the $x$ dimension is generally small, we perform decomposition along the $t$, $y$ and $z$ dimensions. We chose this 3-D decomposition due to its superior performance compared to other possibilities, such as 1-D decomposition along the $t$ dimension. Also, 1-D and 2-D decompositions are more limiting in the amount of available parallelism in the computations compared to a 3-D decomposition. For simplicity, without the loss of generality, we set the $y$ and $z$ dimension sizes of the tile under consideration as $_pn_y = n_y$ and $_pn_z = n_z$.

We follow the CUDA programming paradigm, and define a CUDA thread block in phase 1 as a 3-D array of threads, of size $b_t \times b_y \times b_z$. The number of thread blocks hence generated would be $\left\lceil\frac{N}{b_t}\right\rceil \times \left\lceil\frac{n_y}{b_y}\right\rceil \times \left\lceil\frac{n_z}{b_z}\right\rceil$. Each thread in a thread block is mapped to a set of unique elements in $M_I$ to be computed: thread $T_{i,j,k}$ is responsible for the element tuples $\{q_{x_l}, q_{y_j}, q_{z_k}, t_i\}$, $0 \leq l < n_x$. This mapping can be defined as

$$T_{i,j,k} \xrightarrow{map} (q_x, q_{y_j}, q_{z_k}, t_i), \quad (6)$$

where, $0 \leq i < b_t, 0 \leq j < b_y, 0 \leq k < b_z$. Hence, $T_{i,j,k}$ computes the inner values $F_{t_i}(q_x, q_{y_j}, q_{z_k})$ for all $q_x$. An illustration is shown in Fig. 4.

In phase 2, we follow a similar technique for the sum-reduction, but now we can no longer exploit decomposition along the triangles since this is to be reduced. The computation of $M$ is, thus, decomposed into a *grid* of 3-D *blocks*. A block is sized $b'_x \times b'_y \times b'_z$, and each block corresponds to a CUDA thread block. A thread $T_{i,j,k}$ is mapped to a unique $q$-point. A simple mapping in this case can be

$$T_{i,j,k} \xrightarrow{map} \vec{\mathbf{q}}_{i,j,k} = (q_{x_i}, q_{y_j}, q_{z_k}). \quad (7)$$

Note that in this phase we have included the $x$ dimension in the decomposition. This is to have more flexibility during the computations, and increase the parallelism when possible. In the phase 1, decomposing along $x$ did not have any performance gain, hence for simplicity we did not decompose it. An example of the decomposition and mapping scheme is shown in Fig. 4. Hence, thread $T_{i,j,k}$ is responsible to compute the final form factor value $F(\vec{\mathbf{q}}_{i,j,k})$ by summing up $F_{t_l}(\vec{\mathbf{q}}_{i,j,k})$ over triangles $t_l$, $0 \leq l < N$. At the end of this phase, we obtain the final matrix $M$. One will note that the sizes of these matrices, $M_I$ and $M$, tend to grow rapidly as resolution or number of triangles is increased. A single GPU has limited device memory, and in many typical cases, will not be able to hold these matrices. We tackle this issue next.

### C. Handling Memory Limitations

Large memory requirements necessitate a careful use of the available memory, which is also an essential key to obtaining high-performance. Once more we take the advantage of high data parallelism in the form factor computations.

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 4-D *hyperblocks*, uniquely covering all the $q$-points and triangles. Let us denote a hyperblock by $M_h$, and let its size be $h_x \times h_y \times h_z \times h_t$, $(0 < h_\alpha \leq n_\alpha$, $\alpha \in \{x, y, z, t\})$. For a given hyperblock, its *maximal-set* comprises of all hyperblocks which cover the same $q$-points (but different sets of triangles). Each such maximal set in $M_I$ can be uniquely mapped to a *block* $M_b$, a 3-D sub-matrix of $M$, of size $h_x \times h_y \times h_z$, where the coordinates of the $q$-points in this block are equal to those in the corresponding hyperblocks. This is illustrated in Fig. 4. The total number of such hyperblocks constructed in $M_I$ is, hence, equal to $\left\lceil\frac{n_x}{h_x}\right\rceil\left\lceil\frac{n_y}{h_y}\right\rceil\left\lceil\frac{n_z}{h_z}\right\rceil\left\lceil\frac{N}{h_t}\right\rceil$, and the number of corresponding blocks in $M$ is $\left\lceil\frac{n_x}{h_x}\right\rceil\left\lceil\frac{n_y}{h_y}\right\rceil\left\lceil\frac{n_z}{h_z}\right\rceil$.

The main idea here is to decompose the computations such that each resulting hyperblock can be completely handled in

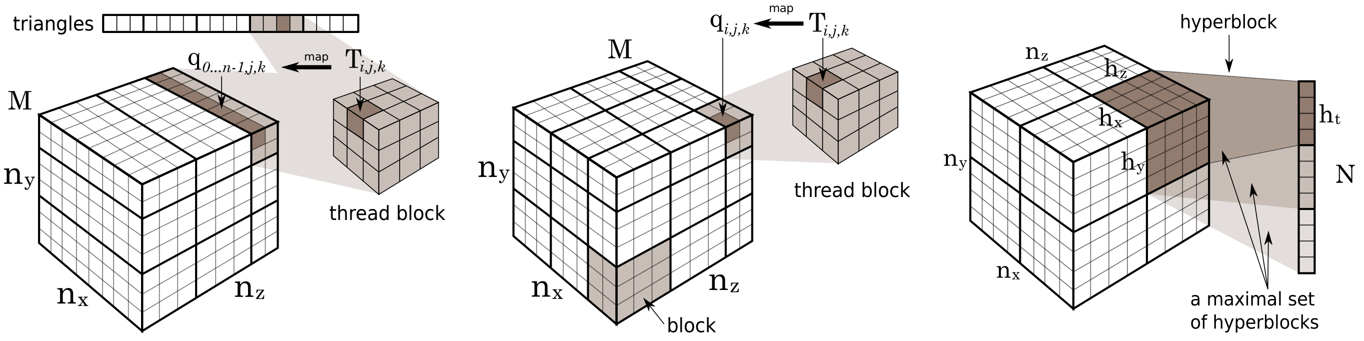

Fig. 4. **(Left)** Phase 1 – Decomposition of computations during the first phase is done along the triangles and $y$, $z$ directions. A triangle is a coordinate in the fourth dimension for all $q$-points in the $Q$-grid. **(Middle)** Phase 2 – Decomposition of $M$ into *blocks*, and mapping of CUDA threads to the $q$-points. Each thread is responsible for the reduction over all the triangles at its mapped $q$-point. **(Right)** 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 available memory at once. Once we decompose the matrix $M_I$ into hyperblocks, the memory requirement to process one hyperblock is $ch_xh_yh_z(h_t+1)$ bytes, where $c$ is a constant representing the number of bytes used to encode a single value. Thus, the size of a hyperblock can be set to fit within the available memory.

We use these hyperblocks as our subproblems to be computed in the limited memory. Hence, we set the size of the input in phase one described earlier by substituting $n_x$ with $h_x$, $n_y$ with $h_y$, $n_z$ with $h_z$, and $N$ with $h_t$. Note that we can easily decompose the computations along the fourth dimension $t$ because summation operation is both associative and commutative. Therefore, the reduction phase needs to be divided into two steps as follows:

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

Phase 1 computation of a hyperblock $M_h$ is immediately followed by phase 2 reduction on this hyperblock. Here the reduction is a partial reduction into a 3-D matrix, $M_p$. A number of iterations would be needed (this number is equal to the total number of hyperblocks) to perform the complete computations. Each iteration consists of computing a hyperblock and generating partially reduced matrix. The number of such partially reduced hyperblocks $M_p$ in a maximal set mapping to one $M_b$ is equal to $\left\lceil \frac{N}{h_t} \right\rceil$. As such, each $M_p$ for the same maximal set is summed after each iteration to maintain the same memory requirement, and construct the final output submatrix $M_b$ in matrix $M$. These operations are carried on by the host CPU simultaneously with computation of next hyperblock on the GPU. We can view this phase of computations as first reducing the size of the fourth dimension from $N$ to $\left\lceil \frac{N}{h_t} \right\rceil$, and then to 1 in order to obtain a 3-dimensional matrix $M_b$.

## D. Algorithm Overview

As a summary of the above descriptions to compute $M$ on a GPU cluster, we present an overview algorithm below summarizing all the computational steps. We also show the use of double buffering in order to overlap computation with memory transfers through *streams* on the GPU.

```
input $Q$-grid: $Q = \{q_{\alpha 0}, \cdots, q_{\alpha(n_\alpha - 1)}\}, \alpha \in \{x, y, z\}$
input Shape triangles: $T = \{t_0, \cdots, t_{N-1}\}$
output Matrix $M_{n_x \times n_y \times n_z}$: $M_{i,j,k} = F(\vec{\mathbf{q}}_{i,j,k})$
procedure FormFactor($Q$, $T$)                          ▷ host code
    Calculate local input $Q$-grid, and $M$.
    Copy local $Q$ and $T$ to device.
    Calculate hyperblock size $h_x \times h_y \times h_z \times h_t$.
    Number of hyperblocks = $\left\lceil \frac{n_x}{h_x} \right\rceil \left\lceil \frac{n_y}{h_y} \right\rceil \left\lceil \frac{n_z}{h_z} \right\rceil \left\lceil \frac{N}{h_t} \right\rceil$.
    Calculate CUDA block size $b_y \times b_z \times b_t$.
    active ← 0.
    for each hyperblock $M_h$ do
        passive ← 1− active.
        if not first iteration then
            Synchronize stream passive.
            Start copy $B_{device}[passive]$ to $B_{host}[passive]$.
        end if
        Launch kernel Phase 1 on stream active.
        Thread $T_{i,j,k}$ executes:
        Start                                          ▷ device code
            for each $x$ do
                $M_h(q_{x,j,k}, t_i) = Fq_{t_i}(q_{x,j,k}) \leftarrow e^{i\vec{q}\cdot\vec{r}} s_{t_i}$.
                Store into $B_{device}[active]$.
            end for
        End
        Calculate CUDA block size $b_x \times b_y \times b_z$.
        Synchronize stream active.
        Launch kernel Phase 2 on stream active.
        Thread $T_{i,j,k}$ executes:
        Start                                          ▷ device code
            $M_p(q_{i,j,k}) \leftarrow \sum_{t_l} M_h(q_{i,j,k}, t_l)$.
        End
        Add $B_{host}[passive]$ to correct location in $M$.
        Synchronize stream active.
        active ← 1− active.
    end for
    Return matrix $M$.
end procedure
```

## V. Optimizing the GPU Code

The performance of the aforementioned procedure is very sensitive to the various decomposition parameters, requiring the search of optimal values for each parameter. Furthermore, developing an efficient and high-performance implementation on the GPUs requires a number of techniques and tricks to optimize both the computations as well as memory accesses and traffic. In this section we will discuss a few examples from such an aspect of our implementation and also our experiences, with a thought that they may provide the reader some insights into GPU code development.

### A. Choosing a Hyperblock Size

Till now we assumed that we are already given the hyperblock size. To start, we will now remove this assumption. One would expect to have the hyperblock size such that it fills the device memory as much as possible, since intuitively this would mean less number of hyperblocks, and hence iterations number of iterations in the algorithm. Also, since the input $Q$-grid and triangle data is accessed multiple times during the computations, having large hyperblock size such that the needed data can fit into the fast memories, would also improve performance. Furthermore, the two phases of the computations derive parallelism from the number of $q$-points and triangles. Too small a hyperblock size would reduce the amount of parallelism available, with smaller number of thread blocks, thereby under-utilizing the multiprocessors.

While on the other hand, after each iteration in the algorithm, the generated partially reduced block $M_p$ is 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 asynchronous data transfers and computations, this step may pose as a bottleneck if the block size is too large, thereby lowering the performance. Also, with a large hyperblock size, the limited caches and shared memory would not be able to hold all the needed data which are frequently accessed. This would increase number of accesses to the slower device memory, reducing performance.

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 be a good balance between the two extremes. In order to demonstrate this, as well as to choose an optimal hyperblock size, we conducted extensive experiments by varying the four parameters $h_x$, $h_y$, $h_z$ and $h_t$, which define the hyperblock size. In the following we show some examples from the results as heat maps. They show snapshots of the execution times with different hyperblock sizes. We use two datasets for these experiments: dataset A with 2,292 triangles, and dataset B with 91,753 triangles, and we use a $Q$-grid of resolution of 3.6M $q$-points as $91 \times 200 \times 200$. Since, $n_x$ is typically small compared to $n_y$ and $n_z$, hence we assign $h_x = n_x = 91$ in these examples. In Fig. 5, we show a heat-map for dataset A (left) and dataset B (right). All the execution times shown are in seconds. We note that we get optimal performances

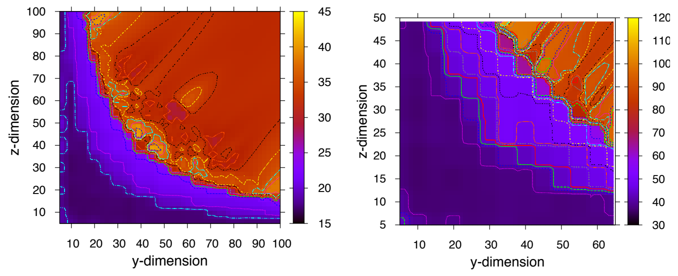

Fig. 5. Execution time heat maps for varying hyperblock sizes on the dataset with $N = 2,292$ (left) and $N = 91,753$ (right). On the $x$-axis is $h_y$, and on $y$-axis is $h_z$. Here, $h_x = 91$ and $h_t = 2000$. The darker/bluer regions are where the best performances are achieved.

towards the lower sizes of $h_y$ and $h_z$, but keeping them too low again increases the runtimes, as can be seen on the lower left corners of the maps. Based on extensive such experiments (also with variable $h_x$ and $h_t$), we selected the hyperblock size parameters $h_x = n_x$, $h_y = 20$, $h_z = 15$, and $h_t = 2,000$ for our further experiments and performance analyses.

### B. Choosing CUDA Thread-block Sizes

With the hyperblock size chosen, we now need to choose the CUDA thread block sizes (and hence, the CUDA thread grid size.) Note that hyperblocks are processed one at a time. As such, the hyperblock size also defines the amount of parallelism available during one iteration. Also, since we have two GPU kernel functions – one for each of the two phases, we need to choose the thread block sizes for both, independently. To avoid redundancy, here we will only discuss the thread block sizes for the phase 1 kernel. Procedures and experiments for phase 2 kernel are similar.

The granularity of scheduling in a GPU is a thread block. It defines the number of threads, and hence, the amount of resources required. Furthermore, the number of thread blocks scheduled to a single multiprocessor also defines the resource divisions (e.g. registers are divided among all the thread blocks). As such, we are again faced with the optimal size values being a good balance between the two extremes. On one hand, more thread blocks per multiprocessor (meaning smaller sizes, given a fixed input) will ensure latency hiding, on the other, they will demand more resources (e.g. number of registers per thread block will be lower, possibly leading to register spilling). Similarly, larger thread blocks demand less resources, share the data copied to the shared memory for each thread block, thereby increasing data reuse from the fast on-chip memory, while they may leave the multiprocessors underutilized. Another factor affecting the choice of these parameters is the warp size. Being SIMD processors, a thread block size as a multiple of warp size will ensure less wastage of resources.

To demonstrate this, we give some examples from our extensive experiments. Again we use similar idea as for the hyperblock size choice. We vary the three parameters $b_x$, $b_y$, and $b_z$ in their possible value ranges (the search space) and

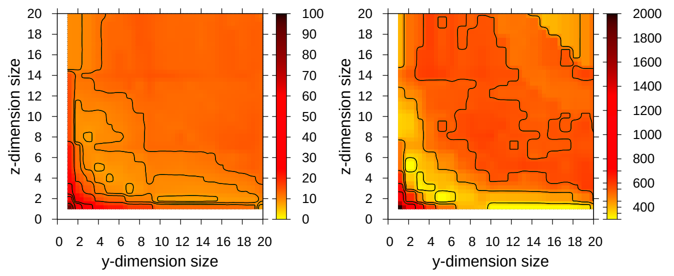

Fig. 6. Execution time heat maps for phase 1 with varying thread block parameter. $b_y$ and $b_z$ are along the $x$- and $y$-axes, respectively. Brighter/yellower regions show the best performances.

obtain the execution times for each. As is clear from the heat maps in Fig. 6, the thread block size may improve/degrade the performance by an order of magnitude.

Through our experiments we selected the thread block size for phase 1 to be $2 \times 4 \times 4$, and for the phase 2 to be $16 \times 2 \times 2$.

It may happen that all these parameters (hyperblock dimensions, thread block sizes for each kernel) are interdependent. To handle such a case to perform a search for optimal parameter values becomes quite hard due to the exponential growth in the size of the search space with the addition of each parameter. One of our next steps in future is to use autotuning, which employs techniques such as branch-and-bound, to address this as well as to choose optimal parameters automatically given a GPU system and inputs.

### C. *Memory Optimizations*

Memory traffic, access patterns and access frequency play an important role in the performance of any application, particularly on specialized processors like GPUs that have a hierarchy of memory from large and slow to small and fast, as well as memories configurable as per the need, and with explicit memory transfers. Major components in the memory hierarchy of a typical compute GPU, from small and fast to large and slow, consists of registers, shared memories, L1 cache, L2 cache, device memory, and host memory.

The computation of a hyperblock in our case proceeds as follows. First, for a thread block the required segments of the $Q$-grid vectors $q_x$, $q_y$, $q_z$, and the triangle definitions are copied from the device memory to temporary buffers in the shared memory by the threads of the thread block. This allows faster access as well as data reuse since entries in each of the transferred segments is accessed multiple times by different threads in the block. The computed values are stored in another buffer in the shared memory, and once the whole block is computed, it is transferred to the device memory.

Data transfers from the global memory to the shared memory is performed as one or more transfers of size 128 bytes. Hence, it is fruitful to encode the to be transferred memory such that it fills 128 bytes segment size as much as possible to reduce bandwidth wastage. As an example, for computation of one thread block of size $2 \times 4 \times 4$ in single precision requires 64 bytes for triangle definitions, 256 bytes for segment of $q_x$ and 16 bytes each for segments of $q_y$ and $q_z$. Properly packing the data into 128 byte segments reduced the number of 128 byte transfers by half from 6, when transferring each data individually, to 3. Furthermore, this method also ensures proper memory coalescing.

Data transfers between the device and host memory have the highest latencies. As one of the basic methods to hide such latencies, we employ double buffering to overlap the transfer of computed and partially reduced output buffers with computation of the next hyperblock. Pinning the host memory buffers ensures efficient transfers between device and host.

Bank conflicts when accessing data in shared memory can degrade the performance. To ensure no conflicts is hence helpful. In our case, as one example, we have 32 threads in a block. Since the number of banks in the shared memory is also 32, we were experiencing high conflicts because each thread was accessing data with a stride of 32 (number of threads), which landed multiple threads to the same bank accessing different words. With a simple change of making the stride to 33, amount of bank conflicts overhead dropped from 24% to just 1.8%. The rest of the bank conflicts were due to access of memory of size 64 bytes by each of the 32 threads. By splitting the access into two steps by letting only even and then odd numbered threads to access the memory, the number of bank conflicts in this kernel went down to 0.

The above optimizations were described in terms of the phase 1 kernel. We used similar techniques to optimize the phase 2 kernel (we will skip the details due to redundancy).

## VI. Parallelization on multi-core CPU clusters

Although GPU clusters prove to be energy efficient, and cheaper than CPU cluster counterparts, general-purpose processor clusters are more common and accessible to larger fraction of the community. Hence, we further extend our codes to work effectively on clusters of multi-core CPUs. Since in the previous sections, a GPU works in conjunction with a CPU, we built upon the same basic framework and replacing the off-loading of computations to GPUs with multi-threaded kernels utilizing all the cores available.

### A. *Across a Multi-core Cluster*

Implementing this code on multi-cores is a lot simpler than on GPUs. Following the same idea, we first decompose the computations in $M$ into a number of equally sized *tiles* along $y$ and $z$ dimensions. The details are the same as covered in Section IV-A. Hence, process $P_{i,j}$ is assigned the tile $M_{i,j}$.

### B. *On a Single Node/Process*

To compute a tile, we again follow similar decomposition procedure as we did for a single GPU. A tile is therefore divided into multiple hyperplocks. This is to ensure constant memory usage during the computations. In a hyperblock, we perform the phase 1 and phase 2 computations. These are performed across the multiple cores available. The phase 1 kernel consists of four nested loops, one each covering the four dimensions. To obtain good performance, we need to be careful about how we order these loops. To preserve locality

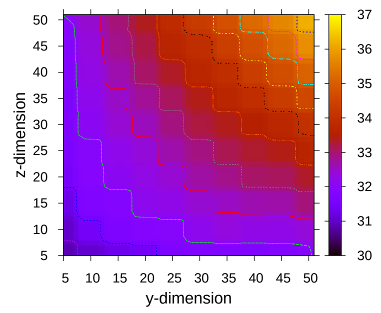

Fig. 7. Execution time heat map for phase 1 on multi-core CPU with varying hyperblock sizes and dataset with $N = 2,292$. On the $x$-axis is $h_y$, and on $y$-axis is $h_z$. Here, $h_x = 91$ and $h_t = 2000$. The darker/bluer regions are where best performances are achieved.

and take advantage of the available caches, we keep the $x$-dimension as the innermost, followed by $y$, $z$ and $t$ in that order, because we use row-major way to store our matrices.

In a typical scenario, the outermost loop, over the triangles, will have the highest loop bound among the four loops (the number of triangles is generally greater than the resolutions along each spatial direction). Based on this fact, and a number of our experiments, we parallelize this outermost loop across the available cores. Hence, each core is assigned a unique set of triangles and is responsible for computing the inner term of form factor for each of the assigned triangles and each $q$-point. This will also result in effective use of caches because each core will be accessing $Q$-grid data in the same order.

We again experiment with various possible hyperblock sizes in order to make a selection of optimal size. An example heat map for this case is given in Fig. 7, where the input consists of 2,292 triangles and $Q$-grid of resolution $91 \times 100 \times 100$. Note that in this case, the variation in execution times is not large, and smaller hyperblock sizes perform slightly better than larger sizes. We attribute this to the large L1, L2 and L3 caches where there are more hits with smaller hyperblock sizes, and number of misses will increase as the hyperblock size grows in relation to these cache sizes.

The reduction kernel for phase 2 is also developed in a similar fashion, but with parallelization across the $y$ (or $z$) dimension. This is because we need to reduce the $t$ dimension, and the size of $x$ dimension is generally small, which would lower amount of parallelism.

Our code is specifically tuned for a Cray XE6 system, consisting of AMD Magny Corus processors. In this system one compute node consists of four sockets, each holding a 6-core processor. This is an example of a NUMA design. To obtain optimal performance, we utilize each processor for a separate parallel task, and hence, generate 6 threads. This configuration performed the best compared to other configurations: 2 parallel tasks with 12 threads each; and, 1 parallel task with 24 threads. We will skip further details on our CPU implementation due to space limitations.

## VII. Analytical Analysis

In this section we will give brief analytical analyses for our GPU and CPU codes. Computational complexity of this problem under consideration is clearly the product of the sizes along all four dimensions: $O(n_x n_y n_z n_t)$. With a naive implementation, the memory requirement would also be of the order of product of the four dimension sizes. Our algorithms make sure that the memory usage remains within the constraints. In fact, computations use a constant size of memory since the requirement is equal to the size of a hyperblock, which once chosen is kept constant, and the output needs to be stored as a $n_x \times n_y \times n_z$ sized matrix.

To gain a deeper insight into the performance capability of the computations under consideration, let us determine the classification of our kernel through its theoretical arithmetic intensity (the ratio flop/byte). On the GPU model, assuming that all the required input is already present in the device memory, there are three main types of read memory transfers: device memory to the multiprocessor (registers), device to shared memory of a multiprocessor, and shared memory to the registers of a multiprocessor. In our case, the first type is not used for any major transfer. Hence, there are two levels of memory traffic during the form factor computations. First let us compute the arithmetic intensity for the case when we ignore the shared memory access latency. Hence, for the optimized phase 1 kernel, the arithmetic intensity is computed to be 2.91. Let us now consider the shared memory access. Assuming that the required data for computation of a block is already in the shared memory, the arithmetic intensity is computed to be 0.97. Hence, during a block computation, there is a good balance of computations and shared memory accesses in the optimum scenario. Poorly optimized kernel, such as one which may have a lot of bank conflicts, will result in degraded performance because the balance will tip towards memory bound. Similar is true for the other way round when arithmetic operations are not optimized.

The theoretical attainable performance of a kernel, according to the Roofline approach of performance modeling, is computed as min{peak performance, peak bandwidth×arithmetic intensity}. On a C2050 GPU, with peak performance of 1.03TFlops and peak bandwidth of 144GB/s, the attainable performance for our phase 1 kernel is 419GFlops, bound by the memory ceiling.

Similarly, for the CPU model, we get an arithmetic intensity of 3.167. On our Cray XE6 Magny Corus platform, the theoretical peak performance is 401.6GFlops, and peak bandwidth is 102.4GB/s. This dictates the maximum attainable performance to be 324.3GFlops, bound by the memory ceiling.

## VIII. Performance Results

The implementation of these codes has been done in C++, along with — on GPU cluster: CUDA 4.2 [6] on the GPUs, OpenMP [7] on the host CPU, and MPI [8] across the nodes; on CPU cluster: MPI for inter-process communication, and OpenMP, with 6 threads per MPI process (at most 4 MPI processes per node). We use the parallel HDF5 [9] binary file format to encode the data defining the input triangles. The output is also stored in the same format, where each process performs parallel I/O operations to write to the output file.

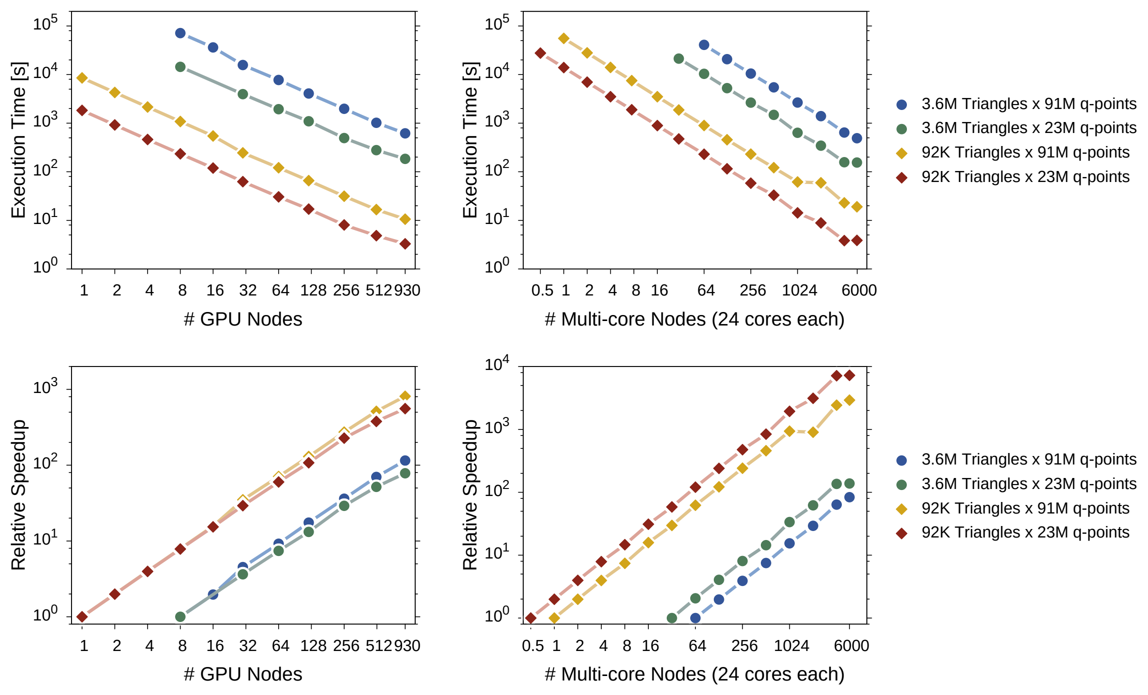

Fig. 8. Strong scaling results for runs on a GPU cluster (left) and CPU cluster (right). The top two graphs show the execution time in seconds taken for four different input configurations. Bottom two graphs show the corresponding relative speedups, w.r.t. the smallest number of nodes which could execute the input cases in reasonable amount of time. Data is shown for up to 930 GPU nodes, and 6,000 multi-core CPU nodes (144,000 cores). The x-axis value of 0.5 nodes in the case of the CPU cluster corresponds to utilizing half a node (12 cores), i.e. running two MPI tasks each with 6 threads.

Using the codes thus implemented, we carried out extensive experiments to analyze their performance. In the following we present some of these results. To start with, we will first describe the configuration of the systems used in our experiments.

We used the GPU cluster *TitanDev*, located at the Oak Ridge Leadership Computing Facility. This developmental cluster consists of NVIDIA Tesla x2050 (Fermi) GPU accelerators, each with 6GB DDR5 device memory and CUDA cores running at frequency of 1.15 GHz, attached to a single AMD Opteron *Interlagos* 16-core CPU with 32GB of DDR3 main memory. This cluster has Gemini interconnects installed. We utilized up to 930 nodes of this cluster. Each GPU ran with a 48KB shared memory configuration.

Recently for a brief period, we also obtained access to 240 nodes of the *Tianhe-1A* GPU cluster, currently ranked 2nd in the top500 list, located at the National Supercomputing Center in Tianjin, China. This system is also built with NVIDIA M2050 Fermi GPUs. We ran some of the scaling experiments on this system and obtained similar scaling as on TitanDev – hence we will omit these results from this paper.

We also used the CPU cluster *Hopper*, located at the National Energy Research Scientific Computing Center in Berkeley. At the time of writing this paper, this system ranked 8th in the top500 list. This is a Cray XE6 system with more than 6,000 compute nodes (we utilized up to 6000 nodes). Each node is a dual AMD Opteron *MagnyCours* 12-core CPU, running at 2.1 GHz. Each node therefore has a total of 24 cores. Each core is equipped with 64KB L1 and 512KB L2 caches. 6 cores share a 6MB L3 cache. Each node has 32GB DDR3 memory, and the nodes are connected with the Gemini interconnects.

In the following experiments, we use two input data-sets: (1) rectangular grating discretized into 91,753 triangles ($\sim$92K), and (2) OPV tomography data discretized into 3,598,351 triangles ($\sim$3.6M). Further, we use two different $Q$-grid resolutions: (1) $91 \times 500 \times 500$ resulting in $\sim$23M $q$-points, and $91 \times 1000 \times 1000$ resulting in 91M $q$-points. These inputs form four different configurations, which we will refer to by 'number of triangles$\times q$-points'. Also keep in mind that all the kernel computations are performed on complex numbers. We use single precision in the following.

In Fig. 8 we show some of the strong scaling results for the GPU and multi-core CPU clusters. We utilized the maximum number of nodes usable on each of the two clusters – 930 GPUs on Titan and 6000 CPU nodes on Hopper. We see that we achieve near perfect scaling in most cases and we believe that our code can easily scale on even larger systems. In Fig. 9 we show scaling results on both clusters for varying input $Q$-grid resolutions, while the number of nodes used and number of input shape triangles is kept constant. And in Fig. 10 we show scaling for varying the shape resolution (number of input triangles) while keeping number of nodes and $Q$-grid resolution constant. In both these scaling results, we again obtain near perfect scaling as expected.

On comparing the execution times on a single node of Hopper and a single node of Titan, it can be seen that a GPU node is generally faster by a factor of about 6.5. While on

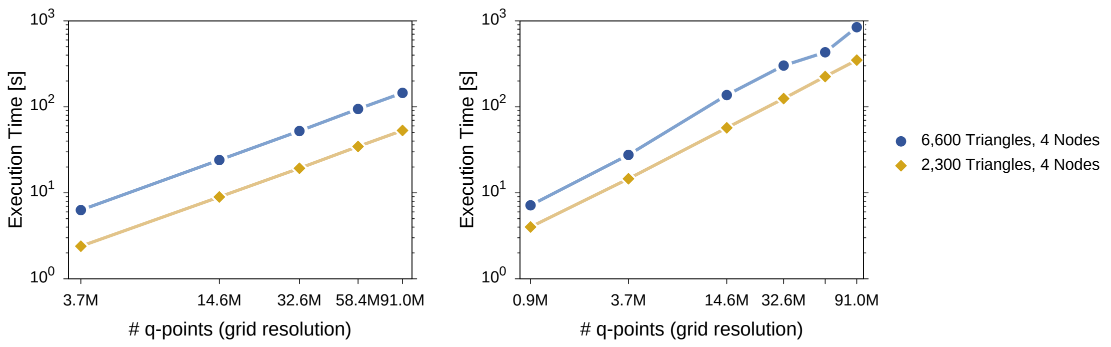

Fig. 9. Scaling on GPU cluster (left) and multi-core CPU cluster (right) w.r.t. varying number of $q$-points in the $Q$-grid. The number of $q$-points represents the grid resolution. Data is shown for resolutions 900,000 up to 91M, and were obtained on 4 nodes on each cluster for two different sized input shape triangle sets.

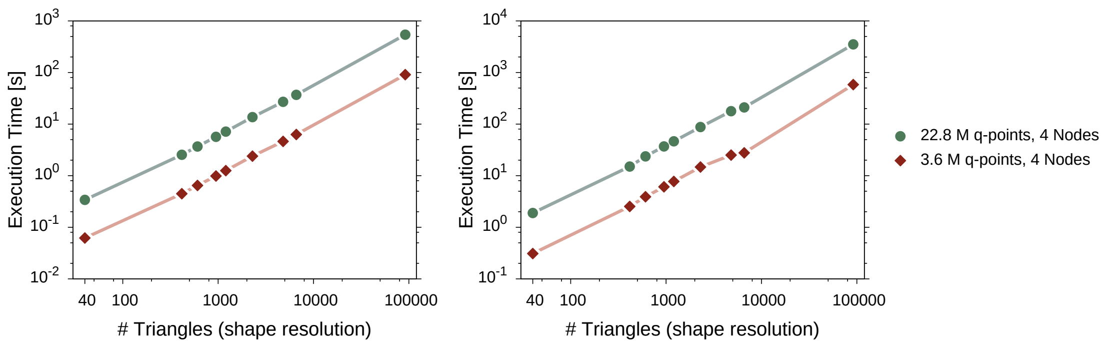

Fig. 10. Scaling on GPU cluster (left) and multi-core CPU cluster (right) with varying number of input triangles. The number of triangles represents the discretization resolution of the shape surface. Data is shown for number of triangles from 40 up to 92K, and were obtained on 4 nodes on each cluster for two different $Q$-grid resolutions.

comparing all 6000 nodes of Hopper against all 930 nodes of Titan, Hopper was faster by a factor of about only 1.3. Although it is not quite fair to compare GPUs with CPUs this way, it just puts the performance into perspective. Our codes obtain 7.12 GFlops on single CPU node and 38.52 GFlops on a single GPU node. Using multiple nodes, 35.824 TFlops are obtained on 930 GPU nodes, and 36.01 TFlops on 6000 CPU nodes. A better measurement of performance would be throughput, in this case defined as the number of points computed per second. On a single CPU node the throughput obtained was 185.97M points/second, while on single GPU node it was 1092.43M points/second. On 930 GPU nodes, maximum throughput obtained was 999.98Billion points/second, and on 6,000 CPU nodes, the maximum was 941.07Billion points/second. Note that the CPU code used above does not take advantage of vector processing. Our codes are still undergoing revisions, and a number of further improvements and optimizations are planned.

## IX. Conclusions

We have designed and implemented 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 speedups of ~125x speedup on one Fermi-GPU card and ~20x on a Cray XE6 24-core node, compared to an optimized sequential CPU code. Further parallelization using MPI led to nearly linear scaling on multi-node clusters. The detailed performance analysis and optimization were presented in the paper. In addition to tremendous runtime reduction, our new codes utilize memory more efficiently, which allows simulations with much larger samples and with higher resolutions than what were previously possible using the old sequential code.

In the future, we plan to use autotuning techniques such as branch-and-bound to aid automatic selection of optimal parameter values, such as hyperblock size and thread block size. In addition to continued optimization of the 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 a GUI input interface and visualization tools. This will allow ALS to truly harness the high-performance computing power.

## Acknowledgments

We thank Samuel Williams for his input on code analysis. We used resources of the National Energy Research Scientific Computing Center, which along with this work is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## References

[1] R. Lazarri, "IsGISAXS: A Program for Grazing-Incidence Small Angle X-Ray Scattering Analysis of Supported Islands," *Journal of Applied Crystallography*, vol. 35, pp. 406–421, 2002.

[2] D. Babonneau, "FitGISAXS: Software Package for Modelling and Analysis of GISAXS Data using IGOR Pro." *Journal of Applied Crystallography*, vol. 43, pp. 929–936, 2010.

[3] "Distributed Data Analysis for Neutron Scattering Experiments," 2010, http://danse.us.

[4] V. Favre-Nicolin, J. Coraux, M.-I. Richard, and H. Renevier, "Fast Computation of Scattering Maps of Nanostructures Using Graphical Processing Units," *Journal of Applied Crystallography*, vol. 44, pp. 635–640, 2011.

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

[6] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide, Version 5.0," 2012.

[7] "OpenMP Application Programming Interface, Version 3.1," Jul. 2011. [Online]. Available: www.openmp.org

[8] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, ser. Version 2.2, Sep. 2009. [Online]. Available: www.mpi-forum.org/docs/docs.html

[9] The HDF Group, "HDF5 User's Guide, Version 1.8.8," Nov. 2011. [Online]. Available: www.hdfgroup.org/hdf5