Tuning HipGISAXS on Multi and Many Core Supercomputers

Abhinav Sarje Xiaoye S. Li
Computational Research Division
Lawrence Berkeley National Laboratory

Alexander Hexemer
Advanced Light Source
Lawrence Berkeley National Laboratory

***Abstract*—With the continual development of multi and many-core architectures, there is a constant need for architecture-specific tuning of application-codes in order to realize high computational performance and energy efficiency, closer to the theoretical peaks of these architectures. In this paper, we present optimization and tuning of HipGISAXS, a parallel X-ray scattering simulation code [1], on various massively-parallel state-of-the-art supercomputers based on multi and many-core processors. In particular, we target clusters of general-purpose multi-cores such as Intel Sandy Bridge and AMD Magny Cours, and many-core accelerators like Nvidia Kepler GPUs and Intel Xeon Phi coprocessors. We present both high-level algorithmic and low-level architecture-aware optimization and tuning methodologies on these platforms. We cover a detailed performance study of our codes on single and multiple nodes of several current top-ranking supercomputers. Additionally, we implement autotuning of many of the algorithmic and optimization parameters for dynamic selection of their optimal values to ensure high-performance and high-efficiency.**

I. Introduction

Multi-core and many-core processors are ubiquitous these days, driving most of the electronics available. These emerging architectures are designed to deliver higher computing power by exploiting multiple levels of parallelism. In high-performance scientific computing (HPC), these architectures play a central role in delivering the much needed compute and memory resources to the wealth of scientific codes and are used on a daily basis. In this paper, we consider one such application code developed by us recently, HipGISAXS, which is a massively parallel X-ray scattering simulation code [1], [2]. This code targets one particular kind of X-ray scattering, called the Grazing Incidence Small-Angle X-ray Scattering (GISAXS). This, available at numerous Synchrotron Light-source facilities, is a widely used tool by scientists for the characterization of macromolecules and nano-particle systems based on their structural properties, such as their shape and size, at the micro/nano-scales. Some of the major applications of these include the characterization of materials for the design and fabrication of energy-relevant nanodevices, such as photo-voltaic cells and energy storage devices, and development of high-density storage media. Although current high-throughput synchrotron light-sources can generate tremendous amounts of raw data at a high rate, analysis of this data for the characterization processes remains the primary bottleneck, demanding large amounts of computational resources. HipGISAXS was developed to address this challenge through the use of massive parallelism.

This X-ray scattering pattern simulation code is based on the Distorted Wave Born Approximation (DWBA) theory, and involves a large number of compute-intensive *form-factor* calculations. Scattered light intensity at a point in inverse space is proportional to the form-factor at that point. A form-factor is computed as an integral over the shape functions of the nanoparticles in a given sample. A simulated sample structure is taken as an input in the form of discretized shape-surfaces, such as a triangulated surface. Intensities are determined at a set of q-points which form a 3D grid. Resolutions of shape-surface discretization, as well as of this spatial 3D grid involved also contribute toward the compute-intensity of the simulations. For computational purposes, the form-factor at a single point is described through a summation over the discretized shape-surface:

$$F(\vec{q}) = -\frac{i}{|\vec{q}|^2} \sum_{t=1}^{n_t} e^{i\vec{q}\cdot\vec{r_t}} q_{n,t} s_t \tag{1}$$

where $\vec{q}$ is a point in the inverse space where light intensity is to be determined, and the summation is over all the n_t triangles of the discretized input sample, $\mathcal{T} = \{t_0 \cdots t_{n_t-1}\}$. Such form factors are computed for all q-points in the 3D grid termed as the Q-grid. We represent it by $\mathcal{Q}$ in the following. $\mathcal{Q}$ is a grid of size $n_x \times n_y \times n_z$ and is described by three vectors, one for each spatial dimension, $q_\alpha = \langle p_0 \cdots p_{n_\alpha-1} \rangle, \alpha \in \{x, y, z\}$. This form factor computational kernel is the focus of our tuning and optimization study in this paper.

Although the various emerging architectures are able to deliver high computational power, they require intensive architecture-aware code tuning in order to do so. This gap between the performance of a straight-forward implementation and that of a highly-optimized code, is described quite well in [3], [4] where they call it the "*ninja gap*". In the work presented in this paper, we address the challenge of adapting HipGISAXS to various parallel architectures through intensive optimizations specific to the systems under consideration. These optimizations involve efficient mapping of computations and data transfers on to a given processor architecture. These architectures have a high-degree of parallelism at various levels ranging from instruction level parallelism to multiple NUMA regions on one processor. Such processors form an integral part of todays supercomputers [5]. Many of these processors are built to be generic enough to be used for general-purpose computing. These are majorly the ubiquitous multi-core CPUs. Some of the processors are designed as highly specialized to deliver computing power for specific types of computations. These processors generally are many-core and typically work

in conjunction with a main general-purpose host CPU. The most common examples of these are graphics processors and Intel's Many Integrated Cores architecture.

Contributions: The goal of HipGISAXS optimizations described in this paper is to take advantage of various high-performance capabilities offered by the different emerging architectures, and make it truly a "high-performance" code. The work presented in our previous paper [1] includes the parallelization and initial implementation of HipGISAXS on Nvidia Fermi GPU clusters, as well as generic multi-core CPU clusters. In our current work presented this paper, we

- implement additional architecture-specific optimizations on Nvidia Fermi and Kepler GPUs;
- parallelize and optimize HipGISAXS on the Intel MIC architecture;
- optimize HipGISAXS on the AMD Magny Cours and Intel Sandy Bridge processors;
- implement auto-tuning of many of the compute and optimization parameters involved; and
- present a detailed performance study on supercomputers based on these four architectures and discuss a comparison.

We conduct our study on the following supercomputers which are based on the above architectures (the rankings below were correct as of the June 2013 Top500 list [5]):

1) *Titan*, ranked 2nd, is a Cray XK7 system which gets most of its performance from the Nvidia Kepler K20X cards available on each node.
2) *Stampede*, ranked 6th, is a cluster with an Intel Phi coprocessor, which are based on the Intel MIC architecture, available on each node.
3) *Hopper*, ranked 24th, is a Cray XE6 system which has dual 12-core AMD Magny Cours processors on each of its compute nodes.
4) *Edison* Phase I, an under development system, is a Cray XC30 system which currently has dual Intel Sandy Bridge processors on each compute node.

II. HipGISAXS Overview

HipGISAXS has been implemented in C++ and uses several capabilities of the C++11 standard. It is a highly modularized code which allows for easy access to various routines for optimization, as well as try and compare different optimizations. Depending on the architecture it is compiled for, it calls the GPU, MIC, AMD, or Intel specific codes for the main computational kernel: the numerical form factor calculations. In the following we give a brief overview of this computational kernel. For more details on its parallelization and implementation, refer to [1].

A. The Kernel

The form factor computation at a single q-point involves accessing data and performing independent calculations for each of the input shape triangles, followed by a reduction over all the triangles. This is done for all the q-points in the problem under consideration. The output matrix $\mathcal{F}$ of size same as the Q-grid $\mathcal{Q}$, is constructed with the results of these computations. This is summarized in the following equation:

$$\mathcal{F} : f(\vec{q}) = -\frac{i}{|\vec{q}|^2} \sum_{t=1}^{n_t} e^{i\vec{q}\cdot\vec{r_t}} q_{n,t} s_t, \quad \forall \vec{q} \in \mathcal{Q}. \tag{2}$$

The computational complexity of this kernel is simply the product of all the four dimensions, $O(n_x n_y n_z n_t)$. A basic implementation of this kernel can be done with four nested loops, with an outer-most loop iterating over all the triangles defining the input shapes and three inner loops iterating over each of the x, y and z dimensions of the q-points, representing the computation over $\mathcal{Q}$. This is then followed by a sum-reduction operation over all the triangles for each q-point. To make referring to these loops easier, let us denote each of them with L_t, L_x, L_y and L_z, respectively. A pseudocode of the kernels is given below. This is how the form factor kernel was originally implemented in HipGISAXS. The definition of a single triangle consists of seven real numbers representing its surface area, three components of its face normal and three components of its centroid. Note that the computations are performed on complex numbers, where re and im represent the real and imaginary parts of a complex number in the following.

```
procedure PHASE1FORMFACTOR(Q, T)
    Input n_x, n_y, n_z, n_t
    Input Q-grid: Q = Q_α{q_α0, ··· , q_α(n_α−1)}, α ∈ {x, y, z}
    Input Shape triangles: T = {t_0, ··· , t_{n_t−1}}
    Output F'_{n_x×n_y×n_z×n_t}
    for each l ∈ {0...(n_t − 1)} do                                  ▷ Loop L_t
        s ← T[l].s                                                   ▷ triangle surface area
        p_x ← T[l].p_x, p_y ← T[l].p_y, p_z ← T[l].p_z               ▷ triangle face normal
        c_x ← T[l].c_x, c_y ← T[l].c_y, c_z ← T[l].c_z               ▷ triangle centroid
        for each k ∈ {0...(n_z − 1)} do                              ▷ Loop L_z
            q_k ← Q_z[k]
            for each j ∈ {0...(n_y − 1)} do                          ▷ Loop L_y
                q_j ← Q_y[j]
                for each i ∈ {0...(n_x − 1)} do                      ▷ Loop L_x
                    q_i ← Q_x[i]
                    q_c ← (c_x q_i + c_y q_j + c_z q_k)/(q_i^2 + q_j^2 + q_k^2)
                    q_p ← p_x q_i + p_y q_j + p_z q_k
                    F'[i, j, k, l] ← s q_p e^{q_c,im}(cos(q_c,re) + i sin(q_c,re))
                end for
            end for
        end for
    end for
end procedure

procedure PHASE2REDUCTION(F')
    Input n_x, n_y, n_z, n_t
    Input F'
    Output F_{n_x×n_y×n_z}
    for each k ∈ {0...(n_z − 1)} do
        for each j ∈ {0...(n_y − 1)} do
            for each i ∈ {0...(n_x − 1)} do
                f ← 0 + i0
                for each l ∈ {0...(n_t − 1)} do
                    f ← f + F'[i, j, k, l]
                end for
                F[i, j, k] ← −if
            end for
        end for
    end for
end procedure
```

B. A Generic Parallelization

Since the problem under study is embarrassingly parallel, a basic parallelization of the code is quite straightforward. A tiling scheme is used to distribute computations among all the available nodes through MPI. On one node, in order to handle

any limitations on the required memory for computations, as well as to increase efficiency, HipGISAXS has a blocking scheme where the computations are decomposed along each of the four dimensions, t, z, y and x, into *hyperblocks*. Computation of hyperblocks is generally performed one after the other. This blocking scheme also has the advantage of making memory transfer and computation overlap, as well as data prefetching, possible. This is specially of essential importance on systems which use an offloading model to transfer computations to a coprocessor. A GPU version is also a part of HipGISAXS, which also incorporates computation decomposition at the level of thread blocks. Figure 1 shows this existing tiling, hyperblocking, and thread blocking schemes.

In the following we present our performance optimizations and tuning, as well as performance analysis, on each of the four architectures under consideration: Nvidia GPUs, Intel Phi coprocessors, AMD Magny Cours CPUs and Intel Sandy Bridge CPUs.

III. Graphics Processors

We start by giving a description of various optimizations performed on HipGISAXS for the Nvidia Fermi and Kepler architecture based GPUs [6]. The starting point for these optimizations is the code developed earlier as mentioned above. In the following, we describe our additional optimizations and autotuning work tailored to the Nvidia GPUs.

A. Nvidia GPU Optimizations

1) Algorithmic Optimizations: For parallelizing the computations on Fermi and Kepler architectures, we encounter a number of choices to make. Each of these choices affects the performance, some greatly and some marginally. Goal of any performance optimization is to make the choices which would give the best performance. On GPUs, one of the first choices to make is to decide which components should define a CUDA thread block. In our case, there are two kernels for each of the two phases of computations: computing the inner term in Eq. 2, followed by a sum reduction along the triangles. In the first phase, the initial implementation of HipGISAXS generated CUDA thread blocks by performing a one-dimensional decomposition along only the t dimension. This clearly limits the amount of parallelism by the number of triangles and disregards the independence of each of the q-points. Hence, it does not expose enough parallelism for cases where $\mathcal{Q}$ is large but the shape definition is small. This kernel was therefore updated to decompose the computations along the three y, z and t dimensions. Recall that the x dimension is typically small, most often being equal to one. This *loop reordering* and *new decomposition scheme* solved the limitations of the initial kernels. To maintain independence in computations, the reduction phase decomposed the domain along the spatial dimensions but not along the triangles since this dimension needs to be reduced. The choice of the sizes of both the hyperblocks and CUDA thread blocks also plays a crucial role in delivering high-performance. We will deal with these choices later in Section III-A4.

One of the main performance disadvantages with the above two kernels is the generation of 4-dimensional intermediate data. Each computed hyperblock is stored by the first kernel in the device memory and then read by reduction kernel. We can eliminate this step of writes and reads by *kernel fusion*. The idea is to keep reducing the computed components and generate only 3-dimensional output which is written to the device memory. This also has the advantage of using less amount of device memory. Parallelization implemented earlier is hence modified: We generate CUDA thread blocks by decomposition along the y and z dimensions and introduce sequentiality along the t dimension. Hence, each thread now executes a loop over the hyperblock triangles and reduces the computed data on the fly. This kernel fusion improved the code's performance on average by about 87%.

2) Memory Optimizations: Computation of each hyperblock generates partial output corresponding to a submatrix of $\mathcal{F}$. This computed data is copied to the host, reduced with previously computed data corresponding to the same submatrix, and placed at its appropriate position in the larger $\mathcal{F}$. This work is performed by the host CPU while the GPU carries on computations on next hyperblock. Double-buffering is commonly used to hide such memory transfer latencies. We explored, going one step ahead, the use of k-buffering. It was interesting to note that while the performance of triple-buffering was around 5% better than double-buffering, the performance of k-buffering for $k \geq 3$ was about the same (we tried until $k = 32$). Since the memory copy latency is much smaller than the compute duration for each hyperblock, once the latency was completely hidden with triple-buffering, increasing the number of buffers did not have any affect. We verified this against profiler trace data.

Usage of the fast on-chip shared memory on each SM of a GPU is critical to obtain good performance. In our case, shared memory can be used to store the input Q-grid and shape triangles information as well as the computed output. Storing input makes data reuse possible, minimizing the amount of memory transfers taking place. Storing the output in shared memory first and then writing to the device memory provides good memory coalescing, increasing the data transfer bandwidth. Storing both input and output on-chip can be expensive since shared memory usage may limit the number of simultaneously scheduled thread blocks on an SM, thereby limiting parallelism and hence, performance. A tradeoff comes into place. After experimentation with the possible cases, our choice was to use shared memory only for the input data: parts of the vectors q_y and q_z defining a particular thread block, and the triangles definition data. All threads in a thread block collectively load the required input to the shared memory in minimum possible number of load operations, and then perform computations on them, storing the results directly into the device memory.

3) Low-Level Optimizations: As described earlier, the definition of a single shape-surface triangle consists of seven real numbers, representing its surface area, three components of its surface normal, and three components of its centroid coordinates. Since this size (7) does not provide efficient memory-alignment, we padded each triangle definition by one more number to make the count as 8 real numbers. In single-precision, this represents 32 bytes. Doing so aligns each triangle definition to a 32-byte line, making memory transfers more efficient. Further, a manual analysis of the PTX code generated by the compiler for the innermost loop L_t of the kernel revealed that the shared memory address calculation

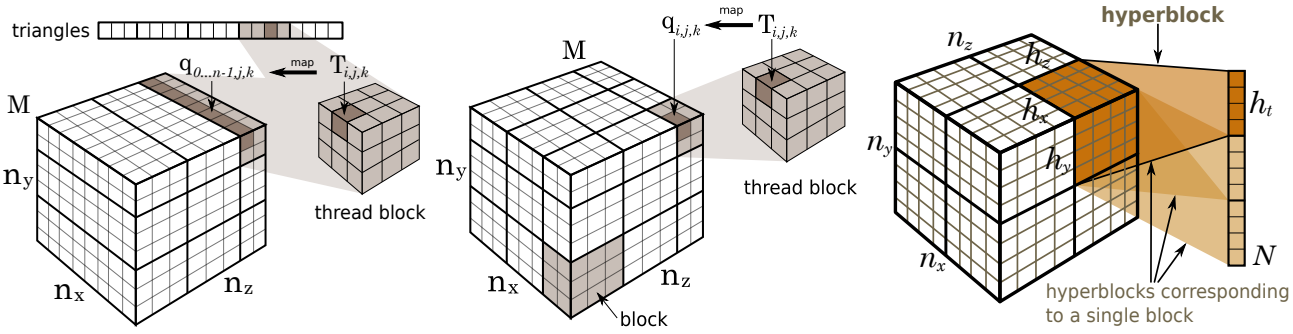

Fig. 1. Problem decomposition schemes. (Left) Phase 1 kernel maps a thread block to a subset of q_y, q_z and $\mathcal{T}$. (Center) Reduction phase kernel maps a thread block to a unique submatrix of $\mathcal{Q}$. (Right) Hyperblocking scheme to handle memory limitations. Graphics taken from [1].

for the 7 components consisted of about one multiply and two adds each. This code is shown below on the left side:

```
mul.wide.u32 %rd35, %r101, 4;
add.s64 %rd37, %rd11, %rd35;
add.s32 %r75, %r101, 1;
mul.wide.u32 %rd38, %r75, 4;
add.s64 %rd39, %rd11, %rd38;
add.s32 %r76, %r101, 2;
mul.wide.u32 %rd40, %r76, 4;
add.s64 %rd41, %rd11, %rd40;
add.s32 %r77, %r101, 3;
mul.wide.u32 %rd42, %r77, 4;
add.s64 %rd43, %rd11, %rd42;
add.s32 %r78, %r101, 4;
mul.wide.u32 %rd44, %r78, 4;
add.s64 %rd45, %rd11, %rd44;
add.s32 %r79, %r101, 5;
mul.wide.u32 %rd46, %r79, 4;
add.s64 %rd47, %rd11, %rd46;
add.s32 %r80, %r101, 6;
mul.wide.u32 %rd48, %r80, 4;
add.s64 %rd49, %rd11, %rd48;
ld.shared.f32 %f72, [%rd37];
ld.shared.f32 %f42, [%rd39];
ld.shared.f32 %f25, [%rd41];
ld.shared.f32 %f34, [%rd43];
ld.shared.f32 %f27, [%rd47];
ld.shared.f32 %f36, [%rd49];
ld.shared.f32 %f39, [%rd45];
```

```
mul.wide.u32 %rd1, %r91, 4;
add.s64 %rd2, %rd1, 4;
add.s64 %rd3, %rd1, 8;
add.s64 %rd4, %rd1, 12;
add.s64 %rd5, %rd1, 16;
add.s64 %rd6, %rd1, 20;
add.s64 %rd7, %rd1, 24;
ld.shared.f32 %f40, [%rd1];
ld.shared.f32 %f41, [%rd2];
ld.shared.f32 %f42, [%rd3];
ld.shared.f32 %f43, [%rd4];
ld.shared.f32 %f44, [%rd5];
ld.shared.f32 %f45, [%rd6];
ld.shared.f32 %f46, [%rd7];
```

In order to reduce the number of operations, and instructions, in L_t, we replaced this code with hand-written PTX code. This code takes advantage of the fact that the memory location offset for each of the 7 triangle components is already known beforehand. Hence, 6 of the 7 multiplies and half of the adds could be eliminated. This code is shown above on the right side.

Since each thread in a thread block loops over all the triangles in the corresponding hyperblock, we can take advantage of loop unrolling. Interestingly, the execution time of the kernel reduced by almost 50% on using an unroll factor of 16. In Figure 2 is shown the trend of execution time (normalized) with the unroll factor.

Nvidia GPUs provide hardware support for single-precision sine and cosine functions at the cost of accuracy. In our optimizations, we take advantage of them since this code does not require high-precision. We implement our own optimized functions for complex number operations and use intrinsics whenever possible to improve performance.

4) Autotuning Decomposition Parameters: In Section III-A1 we talked about various decomposition parameters, such as the hyperblock size and the CUDA block size. Making an optimal choice of these parameters is crucial to obtaining good performance. These parameters

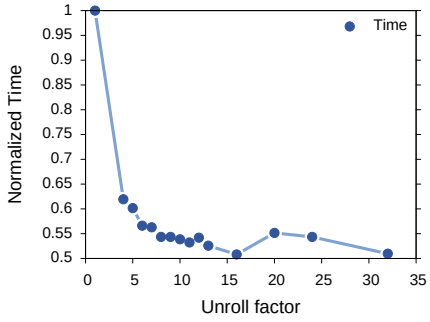

Fig. 2. The effect of loop unrolling on performance. Unroll factor means that the loop is unrolled that many times. Note that best performance is achieved for unroll factors 16 and 32.

are architecture dependent, and are generally independent of the problem instance. Hence, the code needs to be tuned for optimal parameters only once for a given system. In the following, we study the behavior of the execution time with respect to the possible parameter values.

Recall that a hyperblock size is defined primarily by the dimensions y (h_y), z (h_z) and t (h_t). Each parameter has a tradeoff attached to it. For instance, a larger value of h_y (or h_z) means smaller number of resulting hyperblocks. Computation of each such hyperblock would require larger amount of system memory, resulting in larger volume of each memory transfer (although fewer number of transfers). Depending on how much of these data transfer latencies can be hidden with computations dictates the performance. Also, a fewer number of hyperblocks means less overlap for border cases, which might be significant given the large size of the hyperblocks. Similarily, a smaller value of h_y (or h_z) would generate a larger number of smaller hyperblocks. The size of a hyperblock needs to be sufficient to occupy all the SMs on the GPU. Too small a size will not provide enough parallelism and, hence, result in under-utilization of the SMs.

Figure 3 (left) shows an example of the variation in performance as a heat-map with varying values of h_y and h_z, given a constant CUDA thread block size. The first thing we notice in this figure is its banded nature. Also note that the performance is not symmetrical for y and z dimensions. We attribute this primarily to the data organization in memory. We have taken $n_x = 1$, and follow a row-major order storage scheme, which groups the y values together. Each hyperblock is decomposed

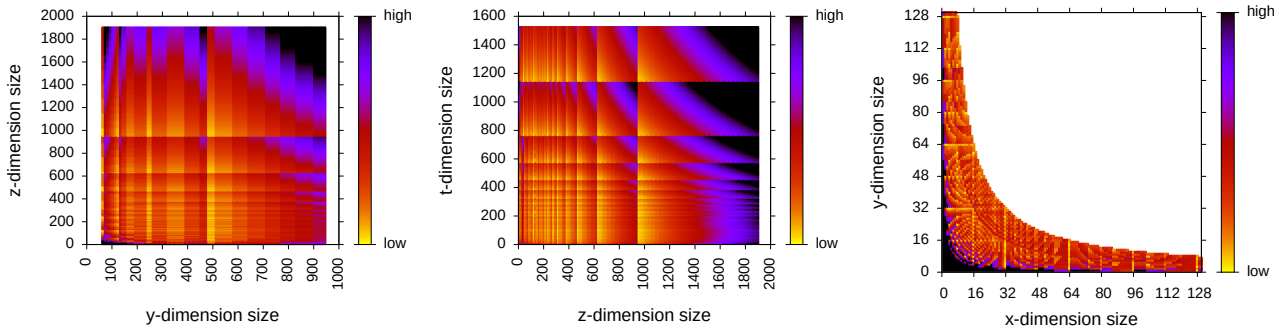

Fig. 3. (Left and Center) The variation in execution time with change in hyperblock sizes. Low is better (yellow regions). (Right) Variation of performance with the choice of CUDA thread block size. The x and y dimensions here correspond to the thread block dimensions.

into thread blocks and each vertical band corresponds to the same number of warps for a given h_z. The change in runtime with h_z is more gradual. Suppose for a given h_z, the matrix $\mathcal{F}$ is perfectly decomposed into equal sized hyperblocks, and we observe good performance. If we increase this h_z by one, an offset in decomposition is created due to which the last set of hyperblocks are partly empty, decreasing performance a little. Further increments of h_z result in this empty region to grow, continually decreasing performance. When h_z reaches a value where $\mathcal{F}$ is perfectly decomposed, we observe a higher performance. This is exactly what we observe in the heat-map. Similarly, Figure 3 (center) shows a heat-map, with varying h_z and h_t values. Increasing h_t creates a similar situation as we saw for h_z above. Hence, this looks more symmetric along h_z and h_t.

The choice of CUDA thread block sizes poses different criteria. Firstly, the total number of threads in a CUDA thread block, i.e. the product of thread block dimension sizes, is limited to 1024 on Fermi and Kepler. Figure 3(right) shows the trend in execution times with respect to variation in the sizes of the two dimensions of a thread block. It can be clearly seen that the sizes which are multiple of 32 give the best performances. This is because a warp size is 32, and a full warp would give better performance due to better utilization than the one which is partially empty.

We implement autotuning in our code which makes an optimal choice of these block size parameters. It is done through an exhaustive search within the parameter space in parallel. As we mentioned above, these parameters are architecture dependent and not problem instance dependent, hence setting them once is enough on a given system. All executions of the code can then use these values, amortizing their auto-tuning computation cost.

B. Theoretical Analysis on GPUs

To compute the number of floating-point operations (FLOPs) on GPUs, we refer to the generated PTX code. We count each of the special functions of reciprocal, exponent and sine-cosine as one floating-point operation each since the hardware provides support for these functions. Each operation on complex numbers is composed of one or more floating-point operations. The main kernel consists of total of 22 FP multiply operations, 3 FP additions, 3 FP subtractions, 9 FP FMA operations, 2 FP reciprocal, 2 FP absolutes, 1 exponent, 1 sine and cosine operations. Of these 1 multiply and 1 FMA lie outside the loop L_t. Hence, we have a total of 42 FLOPs in a single iteration of L_t, giving a total of $42n_t+2$ FLOPs for the form factor computation at one q-point. The total FLOPs in the computation of the kernel among all hyperblocks is therefore $n_x n_y n_z(42n_t + 2)$.

TABLE I. SINGLE NODE EXECUTION TIMES IN SECONDS FOR VARIOUS INPUT SIZES ON M2090 AND K20X GPUS.

$\mathcal{T}$	$\mathcal{Q}$	M2090	GFLOP/s	K20X	GFLOP/s
6,600	2M	0.68	813.6	0.25	2172.2
6,600	8M	2.73	813.2	1.02	2167.2
91,753	2M	9.47	813.4	3.56	2159.1
91,753	8M	37.9	813.4	14.25	2161.5

To compute the arithmetic intensity (flop/byte ratio), let us consider the computation performed by a single CUDA thread block. For ease of representation, let the CUDA thread block dimension sizes be n_x, n_y, n_z and n_t. In the optimized version of the code, there are a total of $4(n_x+n_y+7n_t)+8n_z$ bytes read and $8n_x n_y n_z$ bytes written. Hence, the arithmetic intensity is linear in number of triangles, $O(n_t)$. The kernel is clearly compute bound.

C. Performance Analysis on Titan

In the following, we first cover the performance analysis of our GPU optimized code on both Fermi and Kepler architectures. Specifically, we consider the M2090 and the K20X GPUs. M2090 has a theoretical peak performance of 1331 GFLOP/s in single-precision, while K20X has almost 3950 GFLOP/s. In Table I we show the performance of our code on a single GPU node. Our code obtains about 814 GFLOP/s on the M2090, and about 2,172 GFLOP/s on K20X GPUs. The former is 61.2% of the theoretical peak and the latter is about 55% of the theoretical peak.

We also perform scaling study on the Titan supercomputer, which is a Cray XK7 with an Nvidia Tesla K20Xm available on each of its nodes. In Figure 4 we show the strong scaling of our code for an input data consisting of 2M and 8M q-points and 7.5M triangles defining the shape of an organic photovoltaic

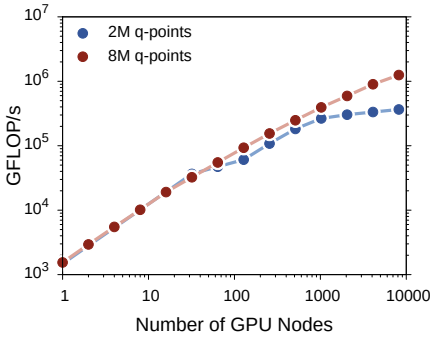

Fig. 4. Strong scaling on the Titan supercomputer. Data is shown for two different Q-grid sizes, and 7.5M triangles. Note how the Q-grid size affects the performance. The code achieves 1.25 PetaFLOPs of sustained performance when using 8,192 GPU nodes.

(OPV) sample. It can be seen that our code scales well. For the case with 2M q points, there is not enough parallelism in the problem instance to efficiently utilize more than 1,024 GPU nodes and, hence, we see flattening in its performance. For the second case with 8M q-points, this is not the case and the code scales up to the available 8,192 GPU nodes. It attains a performance of about 1.25 PetaFLOP/s when using 8,192 nodes.

IV. Many Integrated Cores (MIC)

The second architecture we consider is Intel's Many Integrated Cores (MIC) architecture. Intel Xeon Phi processors [7] are based on this architecture. This platform acts as a coprocessor connected to a host CPU. Although codes can run in native mode on MIC without the intervention of the host CPU, in our case we use the offload model where the host offloads the kernel computations onto the coprocessor. We chose this model due to three major reasons: HipGISAXS uses certain third-party libraries such as imaging and parallel I/O libraries which either are not available for the MIC architecture or perform best on generic multi-core CPUs, the host in our case, since a single core performance of the host CPU is generally higher than single core performance of this many-core coprocessor. Secondly, by offloading the compute intensive kernels to the coprocessor, the host is available to perform other computations simultaneously, which is beneficial in the case of HipGISAXS since it can perform computations such as structure-factor computation independent of the form-factor computations. In addition to these, offloading also enables overlay computations for large problems since the main memory on the coprocessor is limited while the host may have much larger capacity, enabling offloading one hyperblock at a time which would fit into the coprocessor memory.

A. Intel MIC Optimizations

In the following, we present some of our optimizations performed on the form factor kernel of HipGISAXS to adapt it to the Intel MIC architecture.

1) Algorithmic Optimizations: A straightforward parallelization of form factor computations on the MIC can be done by using OpenMP for the nested loops. We are again faced with the choice of which loops to parallelize, and we choose the loops L_y and L_z for this purpose, with loop L_t being the innermost loop executed by all the threads. This choice is beneficial in our case due to two primary reasons. Firstly, it allows better non-local cache utilization by data reuse since each thread requires the shape triangles data. Secondly, it allows for a better vectorization of the code, which we will explain in Section IV-A4. In addition, it eliminates the need for synchronization in reduction operation over the triangles.

We follow a similar scheme as described for GPUs previously for parallelization. Hyperblocking decomposes the computations into smaller subproblems, each of which is computed one at a time using all the available cores on the processor. Since we are using the offload model, this also allows us to exploit data transfer latency hiding through overlap with computations with the use of asynchronous kernel offloading and data copying features available on this coprocessor.

To implement the kernel, we use kernel fusion to merge the reduction step along with the first phase of computing the inner terms.Further, an obvious optimization on the two loops L_y and L_z under consideration is *loop collapsing*. This enables availability of high parallelism for efficient work distribution among the various threads.

2) Memory Optimizations: In implementing the code using the offload model, we minimize the number of data transfers by using the `nocopy`, `alloc_if` and `free_if` features, as well as asynchronous data copy. With these features, we implement triple-buffering scheme for offloading hyperblock computations and computed submatrix data transfers. This enables a perfect data transfer latency hiding, similar to our case for GPUs. All the data buffers used are aligned to 64-bytes (cache line), and padded to maintain this alignment in order to maximize performance.

We perform detailed performance profiling of our code using the Intel VTune Amplifier profiling tool. The obtained data was used to guide our optimizations. One of the major bottlenecks revealed by the profiler was in loading of the shape triangle data. The data in the triangles buffer is originally organized according to the format of the data read from the input shape definition file. This consists of data for each triangle appearing consecutively. As we mentioned previously, this definition consists of 7 real numbers. Hence, the data was stored as such. Since the compiler performed automatic vectorization of the kernel, it had to use gather instructions such as `vgatherdps` to obtain data corresponding to each component of the triangles definition, which had large stride values spanning over multiple triangles. These gather operations are expensive, and proved to be a major bottleneck in the code. Hence, we performed data reorganization to eliminate such operations. While constructing the triangles buffer, we grouped each component of all triangles together. This allowed to have unit stride values for accessing the data to facilitate automatic vectorization. With this optimization we observed a performance improvement of about 33%.

3) Environmental Optimizations: Intel compilers provide a wealth of options to perform optimizations. We took advantage of these, along with those provided by environment variables. The VTune profiler also showed that a significant amount of time was being spent in thread waiting routines of OpenMP. Fortunately, setting the environment

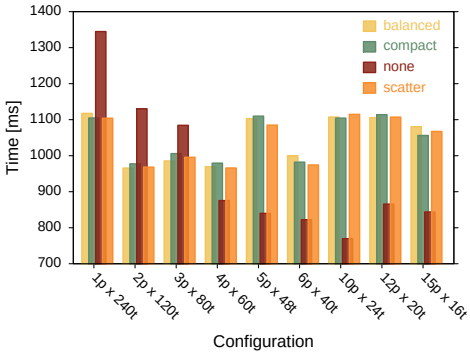

Fig. 5. The performance of various possible configurations on the Intel Phi coprocessor. A configuration is represented by number of MPI processes running on the host (p) and number of OpenMP threads belonging to each MPI process (c) running on the coprocessor. Data is shown for the four different possible affinities: balanced, compact, none and scatter.

variable `KMP_BLOCKTIME` to 0 reduced this overhead. We also employed high pages by setting the value of the environment variable `MIC_USE_2MB_BUFFERS` to 4K. We also set `MIC_KMP_AFFINITY` = `granularity` = `fine`, `compact`, which performed the best among other affinity configurations with our code.

Since we use offload model with MPI processes running on the host, and 240 hardware threads (4 threads per core) on the MIC, it is natural to explore various possible configurations in terms of number of MPI processes running on the host and the number of threads each process creates on the MIC. In our case, the host is a dual 8-core processor providing total of 16 cores. Hence, we explore various configurations ranging from 1 MPI process on the host with 240 OpenMP threads on the coprocessor to 15 MPI processes with 16 OpenMP threads each with different affinities. We employed the variable `MIC_KMP_PLACE_THREADS` to explicitly pin the threads to particular cores depending on which MPI process they belong to. Note that there is no straightforward way to specify the case of 16 MPI processes with 15 OpenMP threads since each core runs 4 threads. The performances we observed for the various configurations using affinities `balanced`, `compact`, `none` and `scatter` are shown in Figure 5. It can be seen that `balanced`, `scatter` and `compact` affinities perform nearly similar. The default affinity of `none` performs erratically: the execution times varied significantly between different runs with the same experimental configuration, sometimes even by 200%. The configurations of 6 MPI processes on host with 40 threads (10 cores) each on the coprocessor and 12 MPI processes on host with 20 threads (5 cores) each on the coprocessor and `compact` affinity are our choices on this platform.

4) Vectorization and Complex Operations: Intel compilers are capable of automatically vectorizing parts of the code in order to utilize the 512-bit vector registers (floating-point and integer). Keeping the loop L_t as innermost, for each thread to execute, is helpful in vectorization since multiple triangles can be processed simultaneously through SIMD parallelism. Analysis of assembly code of the kernel generated by the compiler showed good vectorization, but not perfect. Since most of the computations in our kernel are operations on complex numbers (i.e. each entity consisting of two floating-point numbers), it is not straightforward for the compiler to vectorize them effectively.

The MIC architecture does not support Intel MMX, SSE or AVX vector instruction sets. It implements its own vector instructions (Initial Many Core Instructions [8]) on 512-bit vectors due to the availability of 512-bit integer and floating-point registers. Hence it can perform operations on 16 single-precision or integer data via SIMD. To address the compiler's failure to efficiently vectorize complex number operations, we implemented our own vectorization by using a combination of the low-level vector intrinsics and assembly code. Before we do that, we need to make a choice on how to represent complex number vectors. A straightforward way is to represent one 512-bit vector by 8 single-precision complex numbers, with the real and imaginary components of a complex number stored next to each other. In practice, this approach does not work well in parallel since to perform operations, the real and imaginary components are treated separately and performing operations on interleaved data is not an efficient way to vectorize. Hence, we follow the "structure-of-arrays" schema and define a complex vector by two 512-bit vector components, one for real part and other for the imaginary part as follows:

```
typedef struct {
  __m512 _xvec;
  __m512 _yvec;
} __m512c;
```

Hence, one complex vector holds 16 complex numbers. This way, we can make full use of the SIMD capabilities on MIC. We implement all our operations on vectorized complex numbers using this datatype using the real number intrinsics and instructions. For example, a simple complex-complex multiplication can be written as follows using 4 multiply, 1 add and 1 subtract intrinsics:

```
static inline __m512c _mm512_mul_pc(__m512c a, __m512c b) {
  __m512c vec;
  __m512 t1 = _mm512_mul_ps(a.xvec, b.xvec);
  __m512 t2 = _mm512_mul_ps(a.yvec, b.yvec);
  vec.xvec = _mm512_sub_ps(t1, t2);
  t1 = _mm512_mul_ps(a.xvec, b.yvec);
  t2 = _mm512_mul_ps(a.yvec, b.xvec)
  vec.yvec = _mm512_add_ps(t1, t2);
  return vec;
}
```

We manually vectorize the entire form factor kernel using such intrinsics and assembly code. Since MIC does not provide hardware support for functions such as sine and cosine, they are implemented in software and their vectorized versions are available through libraries such as SVML. We use such functions from SVML in our code. Compared to the compiler generated vectorization, our manual vectorized code showed about 22% performance improvement.

With the above optimizations, the main bottlenecks in our code were now the two SVML functions we used: exponent and sine-cosine. About 80% of the time was spent together in these functions. Hence, we decided to use our own optimized functions for these operations. We obtained a basic implementation of `exp` and `sincos` functions from [9] and performed extensive optimizations (such as instruction re-ordering, arithmetic optimizations, minimize add and multiply instructions through the use of fused-multiply-add). Using these functions in our kernel gave a further overall performance improvement of about 25%. Through micro-benchmarking of

the kernel we computed that our exponent function was about 3.57× faster than the SVML implementation. Similarly, our sine-cosine function was about 1.5× faster than the SVML implementation.

5) Autotuning Decomposition Parameters: Similar to our GPU optimization case, we implement autotuning for optimal choice of the computational decomposition parameters. For the MIC version of the code, we primarily need to tune the hyperblock sizes. An example of the trend of execution time observed for various values of h_y and h_z is shown in Figure 6. Another example with varying values of h_z and h_t is also shown in the same figure.

Quite different to the behavior we saw in the case of GPUs, here all the small hyperblock sizes should be avoided. Although from the figure it is tempting to choose the largest hyperblock size from the top right corner of the plot, it should be noted that the data for these graphs was obtain through extrapolation, and those configurations with large values of all h_y, h_z and h_t are actually not possible due to memory limitations on the coprocessor. Hence, most of the yellow region we see in the figures should be removed, and we should be able to choose parameter values from the center of the heat maps.

Again, the optimal values of these parameters is independent of the problem instance, unless the optimal values are larger than the corresponding sizes in the given problem, the autotuner needs to be executed only once on a system, and all subsequent runs of HipGISAXS will use these values and amortize the cost.

B. Theoretical Analysis on Intel MIC

We refer to our kernel implementation for MIC architecture, done using vector intrinsics and assembly language, to count the number of floating-point operations. In this case we do not consider the special functions as 1 FLOP because they are implemented using the basic arithmetic operations. We do not count floating-point comparisons in our calculations. For operations on complex numbers, we count the number of real FP operations they are built upon. Hence, a real-complex add is just 1 FLOP, complex-complex add is 2 FLOPs, and so on. Our implementation of `exp` contains 26 FLOPs, and `sincos` contains 23 FLOPs. Adding up these and all remaining operations, we obtain $78n_t + 18$ FLOPs for a single q-point, making the overall total to be $n_x n_y n_z(78n_t + 18)$ FLOPs for the entire problem instance. In this case as well, the arithmetic intensity is a function of number of triangles, $O(n_t)$ making this kernel compute bound for MIC architecture also.

C. Performance Analysis on Stampede

We now present some of the performance results we were able to obtain on the Stampede system in the limited time of access and limited number of nodes. We plan to get access to more time and larger number of nodes in order to perform much in depth performance analysis and scaling in the next couple of months. In Table II we list the performance on a single MIC node of the Stampede system with various input sizes. The theoretical peak performance of a MIC card is 2,021 GFLOP/s in single-precision. Our codes were able to achieve 484 GFLOP/s, which is about 24% of the peak.

TABLE II. SINGLE NODE EXECUTION TIMES IN SECONDS FOR VARIOUS INPUT SIZES ON STAMPEDE, CONTAINING THE INTEL PHI COPROCESSORS (MIC).

# Triangles	# q-points	Time [s]	GFLOP/s
6,600	2M	2.27	453.58
6,600	8M	8.769	469.67
91,753	2M	30.767	465.22
91,753	8M	118.49	483.19
7,514,364	2M	2565.18	456.98

In Figure 6, we show strong scaling of our code – on a single node, and across multiple nodes upto 256 nodes of the Stampede system. We were unable to get access to larger number of nodes except for one run on 1,024 nodes of Stampede. It can be seen that the code scales quite well. On 1,024 nodes we ran a problem instance with $n_t = 7,514,364$ and 2M q-points, and the code was able to achieve about 0.1 PetaFLOP/s (1024 nodes).

V. ON MULTI-CORE CPUS

Next we consider general-purpose multi-core CPU architectures for optimizing HipGISAXS. Due to various similarities with previous cases, and space limitations, we will keep this brief.

A. AMD Magny Cours and Intel Sandy Bridge

Cray XE6 and Cray XC30 are built using AMD Magny Cours and Intel Sandy Bridge processors, respectively. These are both general-purpose multi-core processors, the former with 12 cores and the latter with 8 cores. L1, L2 and LLC caches are all system managed. Although it is relatively simple to implement codes for such processors, obtaining high-performance necessitates adapting the code to their specific architectural features. These include effective use of caches through exploitation of spatial and temporal localities in data accesses, and low-level instruction optimizations such as vectorization and ILP exploitation.

We implement data and loop blocking techniques on the innermost loop L_t of our form factor kernel in order to exploit localities on both architectures. We use fused kernel in this case as well, hence, the reduction operation over the triangles minimizes synchronization. Our optimizations of the kernel function on these processors was guided using PAPI [10] through the available hardware counters. AMD Magny Cours provides 128-bit vector registers and supports SSE2 and SSE4a vector instructions. To perform better than the compiler generated vectorization, we implemented the entire form factor kernel using SSE2 vector intrinsics. Intel Sandy Bridge, on the other hand, provides 256-bit vector registers and supports AVX vector instructions. Hence, for this architecture we implemented our kernel using AVX vector intrinsics. In both the cases, we implemented our own optimized versions of operations on complex numbers as well as the special functions `exp` and `sincos`. On Sandy Bridge, since AVX intrinsics do not include integer operations, we implemented them using 128-bit SSE2 vectors. Also, we performed the input shape triangles data reorganization similar to the MIC case explained earlier in order to facilitate efficient vectorization and avoid expensive memory stalls. In addition, this data reorganization includes data blocking to take advantage of localities through available cache as mentioned earlier. Furthermore, we also

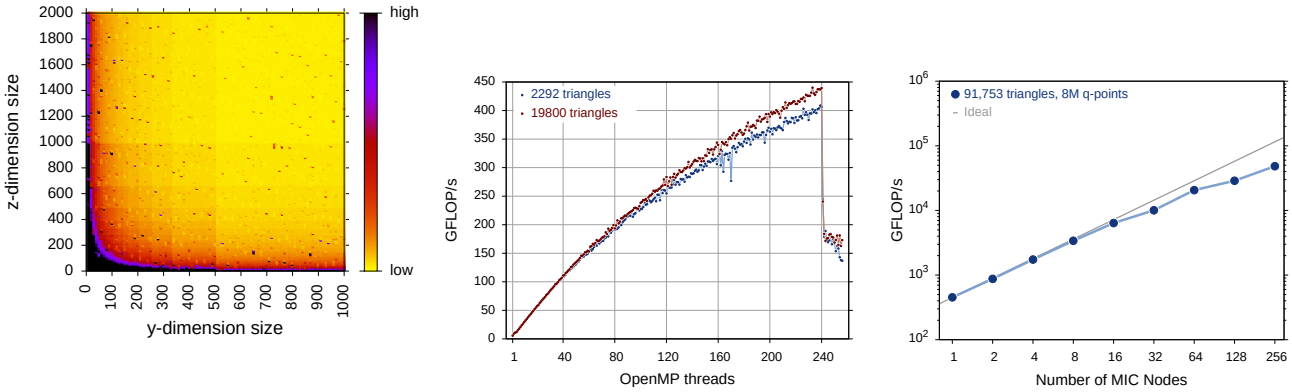

Fig. 6. (Left) Variation in performance with change in hyperblock dimension sizes. Low is better (yellow regions). Note that most of the top right parts are not possible due to memory limitations on the coprocessor. (Center) Strong scaling is shown for a single node, with varying number of threads. (Right) Strong scaling on multiple nodes of Stampede. Each node has one MIC coprocessor. On 1,024 nodes, we achieved 0.1 PetaFLOPs of performance.

unrolled the vector loop by a factor of 2 in order to hide instruction latencies.

Further, we also incorporate autotuning of the hyperblock decomposition parameters into these codes as well. The trend of the execution time with varying hyperblock dimension sizes was quite similar to that we saw for the MIC architecture previously, hence we do now show it here.

B. Theoretical Analysis on Multi-core CPUs

We count the number of floating-point operations in the form factor kernel for our implementations on these multi-core CPUs. These implementations are quite similar (but with different instruction sets: SSE2, AVX). We also count the basic operations for each of the exponent and sine-cosine functions. The total number of floating-point operations in the kernel for a single q-point turns out to be $68n_t+20$ on Magny Cours and $85n_t+16$ on Sandy Bridge. Hence, overall FLOP counts on the two architectures are $n_x n_y n_z(68n_t+20)$ and $n_x n_y n_z(85n_t+16)$, respectively. Again, similar to the case with MIC and GPU implementations, the arithmetic intensity in this case is also a linear function of the number of triangles, $O(n_t)$.

C. Performance Analysis on Hopper and Edison

A single node of Hopper is dual socket containing two AMD Magny Cours processors, providing a total of 24 cores. This XE6 node has 4 NUMA regions, consisting of 6 cores each. In this environment, we have the liberty of choosing the number of MPI processes and the number of OpenMP threads per process while using all the 24 cores. Through experimentation, we determined that for our code, 4 MPI processes with 6 OpenMP threads each on a node performs the best, and we use this configuration in all subsequent experiments on the Hopper system. These processes and threads are pinned to their respective locations in each NUMA region. In contrast, an XC30 node is dual socket with two Sandy Bridge processors, but with only 2 NUMA regions. This processor also supports Hyper-Threading, providing a total of 16 physical and 32 logical cores. On this system, we observed that the configuration consisting of single threaded 16 MPI processes per node, and without Hyper-Threading,

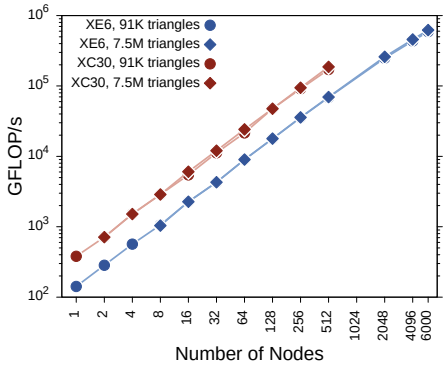

Fig. 7. Strong scaling on the Hopper system, a Cray XE6, and Edison, a Cray XC30. The former is built with AMD Magny Cours processors, each node with 24 cores, making a total of 144,000 cores for 6,000 nodes. The latter consists of Intel Sandy Bridge processors, each node with 16 physical cores, making a total of 8,192 cores for 512 nodes. Our code achieves 0.63 PetaFLOPs on 6,000 nodes of Hopper, and 0.19 PetaFLOPs on 512 nodes of Edison.

TABLE III. SINGLE NODE EXECUTION TIMES IN SECONDS FOR VARIOUS INPUT SIZES (NUMBER OF TRIANGLES AND NUMBER OF q-POINTS) ON AMD MAGNY COURS PROCESSORS OF CRAY XE6 AND INTEL SANDY BRIDGE PROCESSORS OF CRAY XC30, AND CORRESPONDING GFLOP/S.

$\mathcal{T}$	$\mathcal{Q}$	XE6	GFLOP/s	XC30	GFLOP/s
6,600	2M	6.34	141.7	2.97	378.2
6,600	8M	25.28	142.1	11.87	378.3
91,753	2M	88.16	141.7	41.0	379
91,753	8M	352.0	142	164.1	380

performs the best. Hence, we use this configuration for our experiments on the Edison system. In Table III we list single node performance on both systems for several input sizes. The theoretical peak performance of one node of XE6 in single-precision is 403 GFLOP/s, and that of XC30 node is 664 GFLOP/s. With our optimized codes, on a single node of the XE6, we achieved 142 GFLOP/s which is 35.2% of the peak, while on a single node of the XC30, we achieved 380 GFLOP/s which is 57.2% of the peak performance.

Moving on to multiple nodes, in Figure 7 we show the performance scaling of our optimized code on the Hopper and Edison systems. We utilized up to 6,000 nodes on Hopper

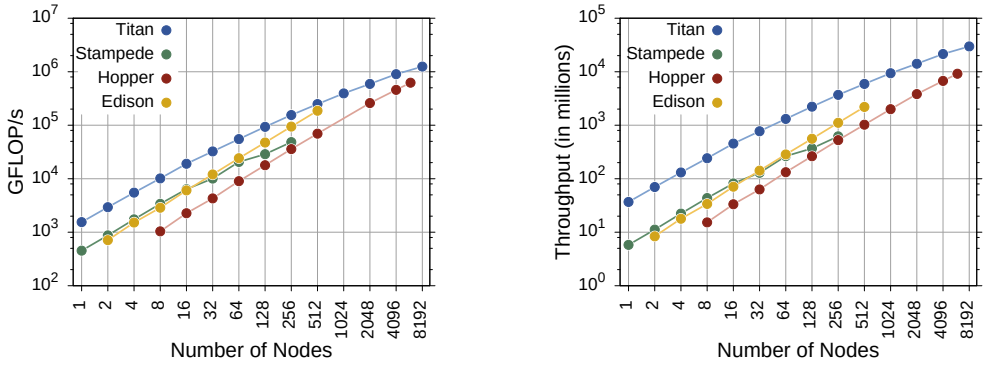

Fig. 8. Comparison of performance of our codes on Titan, Stampede, Hopper and Edison. (Left) Performance in GFLOP/s is shown. (Right) Throughput in TQP/s (number of triangle-q-points processed per second, i.e. one iteration of the loop L_t).

(144,000 cores) reaching 0.63 PetaFLOP/s, and 512 nodes on Edison (8,192 cores) reaching 0.19 PetaFLOP/s.

VI. Comparisons

In the following we discuss a brief comparison of the optimizations and performance of our codes on the different systems presented in this paper. Both Nvidia GPUs and Intel MIC provide a high-degree of fine grained parallelism, which was beneficial in our case due to the independent nature of computations involved in the form factor kernel. Although comparing the different systems is not an easy task, in our case we plot the performances obtained on the three systems on the same number of nodes. Hence, on Titan, one node provides one K20X GPU, on Stampede, one node provides one Intel Phi coprocessor, on Hopper, one node provides dual socket AMD Magny Cours processors with a total of 24 cores, and on Edison, one node is dual socket with Intel Sandy Bridge with a total of 16 physical cores. On a single node of the four systems, our code achieved 142 GFLOP/s on Magny Cours, 380 GFLOP/s on Sandy Bridge, 484 GFLOP/s on MIC and 2172 GFLOP/s on K20X, the building blocks of the four systems. The graphs shown in Figure 8 compare the performance of our codes on all the four systems for comparison, with the same number of nodes on each. It can be seen that the performance of Edison and Stampede are very comparable. Scaling on Hopper and Edison is better than on Titan and Stampede. It should be noted that Nvidia GPUs considered above are the only processors among these which provide hardware support for approximate sine-cosine computations. This made a lot of difference since on other systems, these functions had to be implemented in software using basic mathematical operations.

Although developing a bare-bones working code with acceptable performance is easier on multi-core CPUs like the AMD Magny Cours and Intel Sandy Bridge, than on GPUs or MIC, since they are general-purpose processors, extraction of high-performance from them is no easier than the implementations on graphics processors or the MIC architecture. In an attempt to quantify the effort required by each of these architectures to extract the raw computational power, we note that basic code development person-hours was least for the multi-core CPUs, and highest for GPUs. But taking all together the optimization efforts, the effort went into them is quite comparable. In terms of power, we note that the performance per watt achieved by our codes on each of K20X GPU, MIC, Magny Cours and Sandy Bridge architectures are respectively, 8.98 GFLOP/s/Watt, 1.98 GFLOP/s/Watt, 1.3 GFLOP/s/Watt, and 3.3 GFLOP/s/Watt.

VII. Conclusions

In this paper we described in detail our efforts on optimizing HipGISAXS, a high-performance X-ray scattering simulation code. In our work targeted four specific architectures: Nvidia GPUs (Fermi/Kepler), Intel MIC (Phi processor), AMD Magny Cours and Intel Sandy Bridge CPUs. These different architectures form the basic building block of four of the top supercomputers: Titan, Stampede, Hopper and Edison, respectively. We presented detailed optimization strategies and analyzed the code performance on these supercomputers. On each of these architectures, our code achieved 55% of peak on Nvidia K20X, 24% of peak on Intel MIC, 35.2% of peak on dual AMD Magny Cours, and 57.2% of peak on dual Intel Sandy Bridge. On the scale of the supercomputers used, our code was able to achieve 1.25 PetaFLOP/s on 8,192 nodes of Titan, 0.1 PetaFLOP/s on 1,024 nodes of Stampede, 0.63 PetaFLOP/s on 6,000 nodes of Hopper and 0.19 PetaFLOP/s on 512 nodes of Edison.

VIII. Acknowledgments

The authors thank Samuel Williams for the many helpful discussions. This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, of the Oak Ridge Leadership Computing Facility (OLCF) 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. The authors further acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

[1] A. Sarje, X. Li, S. Chourou, E. Chan, and A. Hexemer, "Massively Parallel X-ray Scattering Simulations," in *Supercomputing (SC'12)*, 2012.

[2] S. Chourou, A. Sarje, X. Li, E. Chan, and A. Hexemer, "HipGISAXS: A High Performance Computing Code for Simulating Grazing Incidence X-Ray Scattering Data," *submitted to the Journal of Applied Crystallography*, 2013.

[3] C. Kim, N. Satish, J. Chhugani *et al.*, "Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology," Tech. Rep., 2011.

[4] N. Satish, C. Kim, J. Chhugani *et al.*, "Can traditional programming bridge the Ninja performance gap for parallel computing applications?" *SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 440–451, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2366231.2337210

[5] "Top500 Supercomputers," June 2013. [Online]. Available: http://www.top500.org

[6] "Tesla Kepler GPU Accelerators," Datasheet, Nvidia Corp., 2012.

[7] "Intel Xeon Phi Coprocessor. Developer's Quick Start Guide. Version 1.5," White Paper, Intel Corp., 2013.

[8] *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, Intel Corp., September 2012.

[9] J. Pommier, "SIMD implementation of sin, cos, exp and log," Tech. Rep., 2007. [Online]. Available: http://gruntthepeon.free.fr/ssemath

[10] "Performance Application Programming Interface (PAPI)," 2013. [Online]. Available: http://icl.cs.utk.edu/papi