Real Time Global Illumination Solutions to the Radiosity Algorithm using Hybrid CPU/GPU Nodes

REU Site: Interdisciplinary Program in High Performance Computing

Oluwapelumi Adenikinju[1], Julian Gilyard[2], Joshua Massey[1], Thomas Stitt[3],
Graduate asisstants: Jonathan Graf[4], Xuan Huang[4], Samuel Khuvis[4],
Faculty mentor: Matthias K. Gobbert[4],
Clients: Yu Wang[1], Marc Olano[1]

[1]Department of Computer Science and Electrical Engineering, UMBC
[2]Department of Computer Science, Wake Forest University
[3]Department of Computer Science and Engineering, Pennsylvania State University
[4]Department of Mathematics and Statistics, UMBC

Technical Report HPCF–2014–15, www.umbc.edu/hpcf > Publications

Abstract

We investigate high performance solutions to the global illumination problem in computer graphics. An existing CPU serial implementation using the radiosity method is given as the performance baseline where a scene and corresponding form-factor coefficients are provided. The initial computational radiosity solver uses the classical Jacobi method as an iterative approach to solving the radiosity linear system. We add the option of using the modern BiCG-STAB method with the aim of reduced runtime through a reduction in iteration count with respect to Jacobi for complex problems. It is found that for the test scenes used, the convergence complexity was not great enough to take advantage of mathematical reformulation through BiCG-STAB. Single-node parallelization techniques are implemented through OpenMP-based threading, GPU-offloading, and hybrid threading/GPU offloading and it is seen that in general OpenMP is optimal by requiring no expense. Finally, we investigate the non-standard-array storage style of the system to determine whether storage through arrays of structures or structures of arrays results in better performance. We find that the usage of arrays of structures in conjunction with OpenMP results in the best performance except for small scene sizes where CUDA shows the minimal runtime.

1 Introduction

We investigate optimization of existing software through parallel execution techniques and mathematical reformulations in the global-illumination computer graphics problem. The radiosity method is given as the solution approach to solving the global illumination problem, where a 3D scene with diffusive surfaces and its corresponding form-factor coefficients are prepared as known variables and used in our parallel solutions.

Given the cutting edge and hybrid cluster here at UMBC, we test different parallel computing approaches including MPI, GPU offloading with CUDA (Compute Unified Device Architecture), and multi-threading with OpenMP (Open Multi-Processing). The compute nodes used for code implementation and timing trials are located in the High Performance Computing Facility (HPCF) at UMBC. Released in Summer 2014, the current machine in

HPCF is the 240-node distributed-memory cluster maya. The newest components of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. These nodes include 19 hybrid nodes with two state-of-the-art NVIDIA K20 GPUs with 2496 computational cores designed for scientific computing as well as 19 hybrid nodes with two cutting-edge 60-core Intel Phi 5110P accelerators. These nodes are connected by a high-speed quad-data rate (QDR) InfiniBand network to a central storage of more than 750 TB.

We identified the computationally intensive portion of the program by evaluating the execution time for each computation stage and processing multiple test scenes using the CPU radiosity implementation. Instead of using a Jacobi method with a fixed number of iterations, we propose to use a modern iterative linear solver as a optimization to the numerical solution. The BiCG-STAB method is chosen given the characteristics of the linear system in question to decrease the iteration count needed for below tolerance solutions to complex systems. Hybrid parallel computing approaches are adopted to significantly improve performance of the solver. Specifically, we introduce a hybrid solution by fully utilizing dual-socket multi-core nodes available through multi-threading techniques with the help of the OpenMP API, and exploit access to massively parallel hardware through GPU-offloading with CUDA. The combination of GPU-offloading and CPU-threading is explored through a hybrid CPU/GPU compute implementation.

Performance metrics are executed and are compared through scalability studies and absolute runtime results. By varying the patch count and scene complexity, we investigate memory allocation and transfer times and the utility of mathematical reformulations.

To motivate the use of GPUs, we actually start by exploring the classical test problem of solving the Poisson equation by the conjugate gradient method. Specifically, we compare the performance of the code for using either several nodes and processes with MPI to using one GPU with cuBLAS.

The outline of the report is the following. Section 2 is devoted to explanation of background and problem statement. We will first introduce the Poisson equation as motivational problem, then discuss the radiosity method. An in-depth description of the open-source package RRV (Radiosity Renderer and Visualizer) follows. Section 3 will explain the methods used and parallel implementation. First we solve the Poisson equation using different methods, including MPI approach, GPU offloading with CUDA, and multi-threading with OpenMP. Then we give a detailed examination of the radiosity computation provided by RRV. We identify the computationaly intensive portion of the program, then solve it with GPU offloading with CUDA and multi-threading with OpenMP. In Section 4, it will be demonstrated that our approach with GPU offloading and OpenMP greatly speed up the Poisson equation and radiosity calculations. Results using array of structures and structure of arrays will be discussed respectively. Section 5 summarizes our conclusions and motivates future work.

2 Background and Problem Statements

2.1 The Poisson Equation as Motivational Problem

We begin with the Poisson equation as a motivational problem, since it provides insights to the performance of linear solvers, which can later be used in the radiosity problem. We want to determine the run times for the Poisson equation over multiple sets of processes and nodes. In theory, for every increase in p processes, the time should decrease by a factor of p. In order to perform performance tests we use the Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{2.1}$$

on the domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$ where $\partial\Omega$ is the boundary of the domain. In order to solve the problem numerically the domain was discretized into $(N+2)$ points in each dimension with spacing $h = 1/(N+1)$ where N is the mesh size. Equation (2.1) was approximated

$$\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial {x_{k_1}}^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial {x_{k_2}}^2} \approx \frac{-u_{k_1-1,k_2} + 2u_{k_1,k_2} - u_{k_1+1,k_2}}{h^2} + \frac{-u_{k_1,k_2-1} + 2u_{k_1,k_2} - u_{k_1,k_2+1}}{h^2} \tag{2.2}$$

for $k_i = 1, 2, ..., N$ and $i = 1, 2$. With these N^2 unknowns we can create a linear system of N^2 equations for the finite difference approximation of (2.1) and following [4] we can use the conjugate gradient (CG) method to produce a solution with a given ϵ error bound. An explicit matrix is also not needed as the matrix-vector product for updating just requires application of the approximation in (2.1).

2.2 Radiosity

Global illumination problems are omnipresent in computer graphics. Early methods such as ray tracing produced visually pleasing renderings but suffered from a need to recompute the solution for any change of view point. The radiosity method introduced by Cindy Goral et al. [2] removed the need to recompute environmental lighting during a perspective change and allowed illumination of a statically lit scene to be computed once for any viewpoint. The radiosity method solves for the steady-state energy (light) distribution in an environment. One limitation of radiosity method is that all reflection is assumed to be diffuse. In other words, the energy (light) flux given off from the surface is equal in all directions. This diffuse reflection assumption limits the radiosity equation's application to only matte surfaces.

Radiosity is defined as the total energy given off from a surface (the sum of emitted and reflected energy) and is given by

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{i,j} \qquad \text{for } i = 1, 2, \ldots, N, \tag{2.3}$$

where

- B_i (radiosity) is the total energy leaving the surface (radiosity) of the ith patch (energy/unit time/unit area),
- E_i (emission rate) is the rate of energy leaving the ith patch (energy/unit time/unit area),
- ρ_i (reflectivity) is the reflectivity of the ith patch (unitless). The reflectivity depends on the wavelength of light,
- $F_{i,j}$ (form factor) is the fraction of energy emitted from patch j that reaches patch i (unitless), and
- N is the number of patches.

When a given system is discretized into N patches and realizing that $F_{i,j} = 0$ a system of linear equations results that can be written

$$\begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix} + \begin{bmatrix} 0 & \rho_1 F_{1,2} & \cdots & \rho_1 F_{1,N} \\ \rho_2 F_{2,1} & 0 & \cdots & \rho_2 F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_N F_{N,1} & \rho_N F_{N,2} & \cdots & 0 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} \tag{2.4}$$

In order to solve this system to acceptable values of $B_1, B_2, \ldots, B_N$ we solve the system for steady-state

$$\begin{bmatrix} 1 & -\rho_1 F_{1,2} & \cdots & -\rho_1 F_{1,N} \\ -\rho_2 F_{2,1} & 1 & \cdots & -\rho_2 F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_N F_{N,1} & -\rho_N F_{N,2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix} \tag{2.5}$$

which can be written as a linear system

$$Ab = e \tag{2.6}$$

with

$$A = \begin{bmatrix} 1 & -\rho_1 F_{1,2} & \cdots & -\rho_1 F_{1,N} \\ -\rho_2 F_{2,1} & 1 & \cdots & -\rho_2 F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_N F_{N,1} & -\rho_N F_{N,2} & \cdots & 1 \end{bmatrix}, \quad b = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix}, \quad e = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix}.$$

2.3 Form Factors

The form factor $F_{i,j}$ is defined as the fraction of energy emitted from A_j that is incident on A_i (Figure 2.1). The sum $\sum_{i=1}^{N} F_{i,j}$ is defined to be 1. The form factor ($F_{i,j}$) is given as

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos(\phi_i)\cos(\phi_j)}{\pi r^2} \, dA_j \, dA_i. \tag{2.7}$$

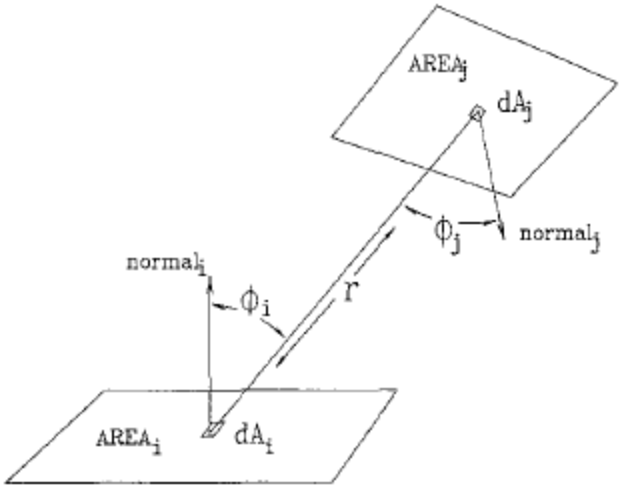

Figure 2.1: Illustration of the form-factor geometry [3]. Used under ACM Non-Commercial Permissions.

Geometrically speaking, the form factor is the fraction of a circular patch i covered by a projection from patch j onto hemisphere around i followed by a orthogonal projection onto the plane of i.

Equation (2.7) does not take into account a patch-to-patch projection that is partially or fully blocked by an intermediate object. A term $HID_{i,j}$ is added to (2.7) to take this into account to yield

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos(\phi_i)\cos(\phi_j)}{\pi r^2} HID_{i,j}\, dA_j\, dA_i. \tag{2.8}$$

2.4 Method of Hemicubes

In order to numerically solve for the form factor $F_{i,j}$, hemicubes [3] are used in (2.8) by replacing the idea of a hemisphere with a cube known as a hemicube. A patch i is chosen and the origin is chosen to be the center of the patch with the positive z-direction chosen to be normal to the plane of patch. A half cube, or hemicube, is centered on the patch (Figure 2.2) and the hemicube is discretized to a desired mesh size. All other environment patches are projected onto the hemicube at i with each mesh point having one corresponding environmental patch. The form factor at patch i is the sum of all appropriately weighted mesh points. A new patch is then chosen and the process repeats until all form-factors are calculated.

2.5 Radiosity Renderer and Visualizer

The open-source package RRV (Radiosity Renderer and Visualizer) [1] is a global-illumination solving and visualizing suite written in C++ and OpenGL. The radiosity computation engine uses a Jacobi Iterative Method with a fixed number of iterations. The `RRV-compute` program is used in conjunction with an .xml scene description format of the geometric components (i.e., primitives, such as polygons) that make up the scene, to compute the

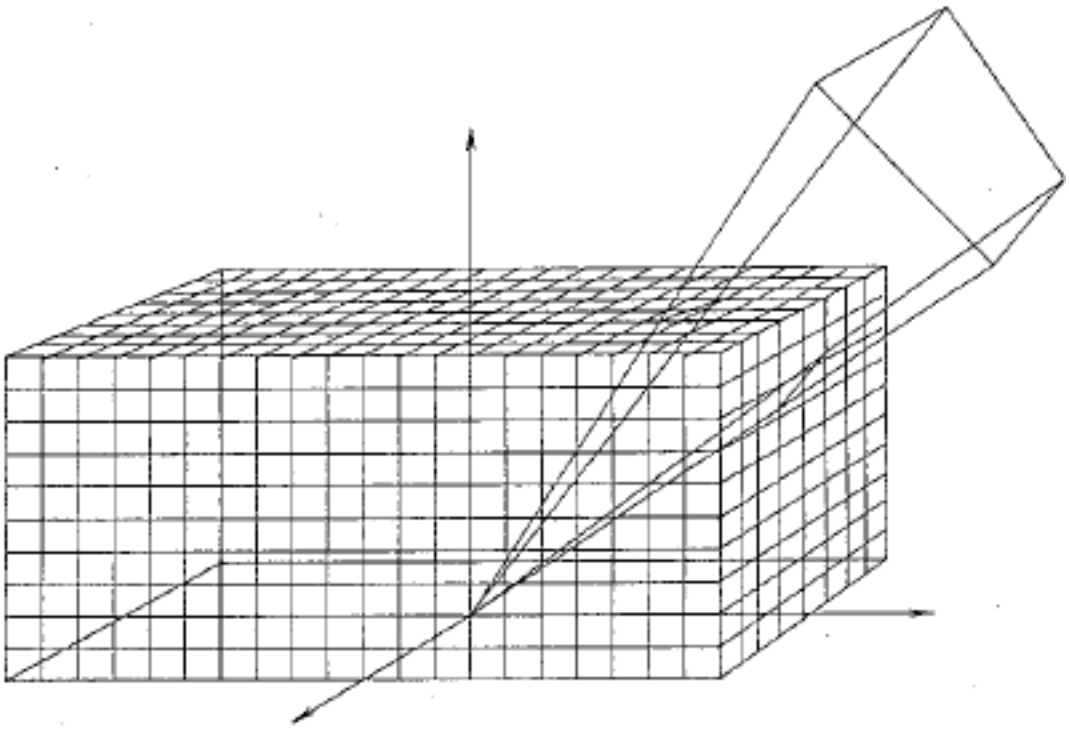

Figure 2.2: Illustration of the hemicube [3]. Used under ACM Non-Commercial Permissions.

global illumination for visualization with `RRV-visualize`. The radiosity algorithm solving `RRV-compute` program is the focus. The source code for RRV is available for download at http://dudka.cz/rrv.

3 Methodology and Parallel Implementation

3.1 MPI Poisson

Distributed-memory parallelization on the CG method was accomplished through the use of a parallel dot-product, parallel vector-scaling, and parallel matrix-vector-product. The $N \times N$ problems are divided evenly among the p processors available to the task. In terms of the discretization space the problem is divided along the x_2 (y) axis, which translates to contiguous memory arrays for each process, but since we use a 5-point stencil process-boundary points need information from another process's memory and thus process-to-process communication is needed between adjacent processes. This was accomplished with point-to-point communication with the `MPI_Isend` and `MPI_Irecv` or `MPI_Send` and `MPI_Recv`, respectively [6].

3.2 CUDA Poisson

Given the complexity of the Poisson equation, executing this problem over the GPUs was the next logical step. GPUs can handle larger problems where $N \times N$ meshes increase, we

would observe a larger speed up vs. using CPU code. GPUs allow for thousands of the same processes to be executed simultaneously in order to complete redundant tasks quickly. In this case, the scenario of vector multiplication is redundant, therefore we can parallelize it across a given number of processes. The basic idea is that a Kernel (CUDA jargon for function) is created by passing through a number of blocks per grid and threads per block, see (Figure 3.1). A block is a virtual arrangement of threads and can be arranged in dimensions between 1 and 1024, the maximum number of threads per block. For example a block could be arranged in the manner like $(32, 32, 1)$, having 1024 threads each with respective x and y coordinates. Additionally it could be arranged $(1024, 1, 1)$ or in any other arrangements, as long as the blocks do not exceed 1024 threads. Each thread then executes the kernel code. Grids therein are composed of blocks but grid arrangements must be arranged in a 2-D or 1-D structure [5]. So in theory one could have a 2×2 grid containing blocks of size $(32, 32, 1)$. This scenario produces 4096 threads to execute the kernel. Arrangements can vary based upon different requirements. Note that it is illogical to make a non-square block arrangement of threads. If one were to make a block arrangement of $(16, 15, 1)$ CUDA would automatically make enough threads for a $(16, 16, 1)$ block arrangement. In this situation, 16 threads are left idle and wasted. With our code, we used a $(16, 16, 1)$ block arrangement and created a grid dynamically by dividing the $N \times N$ size of the matrix by the size of our blocks. Thus in this arrangement we are able to accommodate two-dimensional square mesh sizes. For example, in a 1024×1024 mesh size, we would divide the mesh size by 16 and then we would have a grid of 64 $(16, 16, 1)$ blocks. We arrange ours in 1-D, but the implementation could have been executed in 2-D. Different block sizes could have been used, however, the 16×16 block size was optimal for our purposes.

CUDA is NVIDIA's C-based library designed for leveraging the massively parallel architecture of NVIDIA based GPUs. With our CUDA implementation of C code, we observed considerably faster times over CPU times of one node. We used one node as we thought that processes 1 to 16 would provide the most equitable comparison to the GPU (Graphics Processing Unit). The GPUs that we used for are test were Tesla K20 NVIDIA GPUs with 5 GB of shared memory. Because of the memory limit regarding space of the GPU, we were forced to limit our comparison test to mesh sizes of 8192×8192. For our CUDA implementations, we ran two separate test. The cuBLAS 1 run time utilizes only one cuBLAS function; the `cuBLASDdot` function. We were forced to use the `cuBLASDdot` function because CUDA does not have an atomic add function for double precision. For comparison, in cuBLAS 2, we used only cuBLAS functions. Here we used the `cuBLASDaxpy` function instead of our `axby` kernel. Our `axpby` out performed full cuBLAS functions for all $N \times N$ meshes that we computed. The largest causes for this seems to stem from the function `cuBLASDscale` which scales the vector,but requires a considerable amount of time.

3.3 OpenMP Poisson

OpenMP (Open Multi-processing) is an API (application programming interface) for efficient and cross-platform shared-memory parallelization. So-called shared memory is memory that is shared across all workers (threads). Threads can also have private memory if a variable is marked thread-private with a specific flag. Parallelization is accomplished via `#pragma omp` declarations followed by specific parallel specifications. OpenMP also supports `simd` decla-

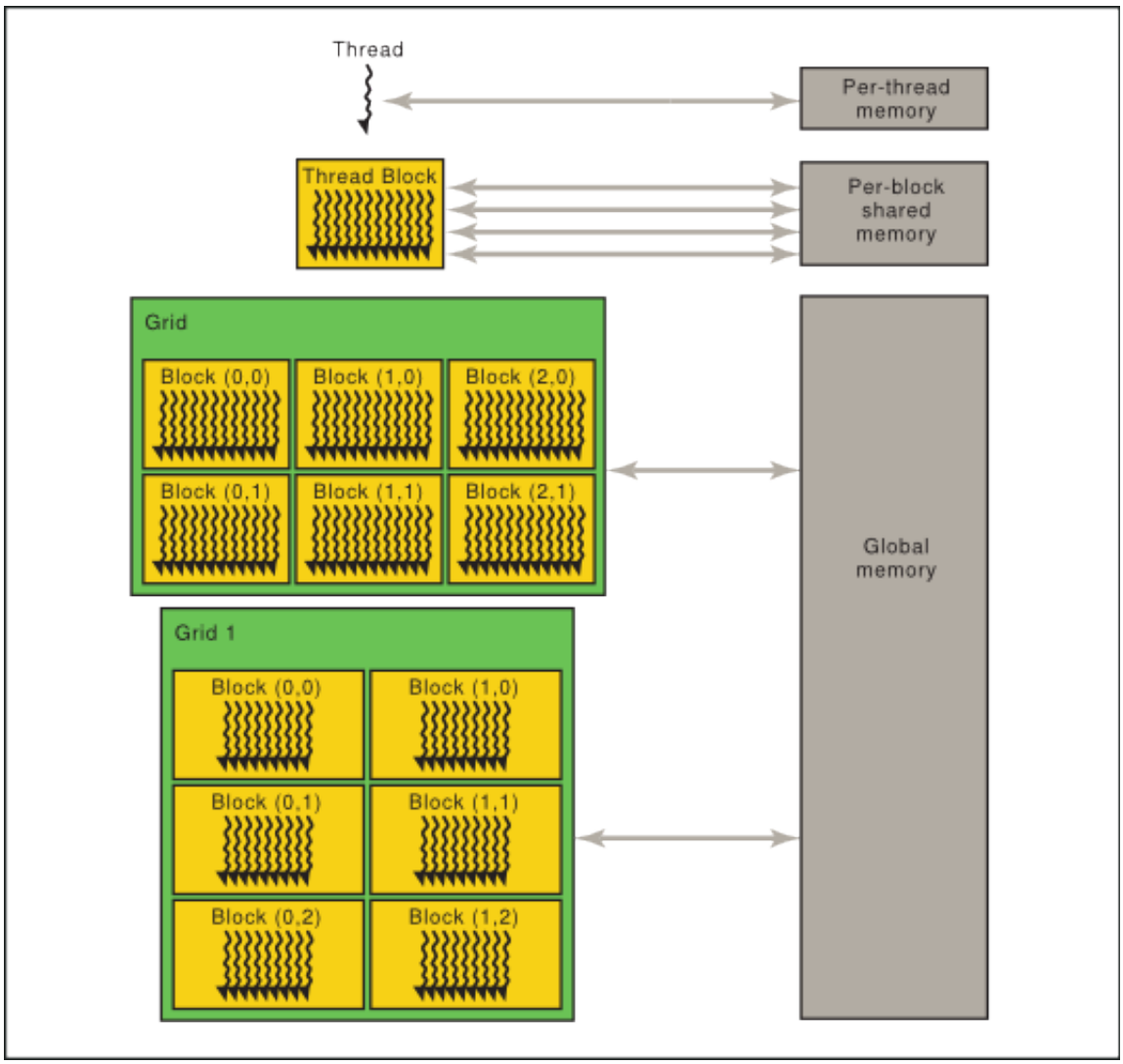

Figure 3.1: Breakdown of the Memory. Provided by NVIDIA [7].

rations for specific vectorization of functions or loops. The loop-dominate functions in the conjugate gradient method were parallelized using these OpenMP declarations.

3.4 Workflow for RRV

The source code for RRV was investigated prior to attempts at parallelization. The RRV work flow is given as follows:

1. Given an appropriately scene description file (in .xml format), a `Scene` object is created and the `Scene` class's `load` function is called on the given scene.

2. During the `load` call an `EntitySet` is populated from the environment descriptors in the input scene description.

3. The `applyEmission` function is called on our `Scene` instance which sets a patch's

radiosity value to its emission value if the emission value is greater. This step allows for faster convergence.

4. `Scene` instance is divided such that triangles are at a maximum of size `scale` which has a default value of 1 but can be set by the `--scale` input flag.

5. A `RadiosityRenderer` is created from the `Scene` object with given steps, form-factor threshold, and maximum cache size. This creates a `PatchSequenceEnumerator`, which is a wrapped `std::vector<>` object, that holds the `Triangle` objects that represent the input file.

6. The `compute` function is called on our `RadiosityRenderer` object. This function does the radiosity computation via (2.4) and runs for `stepCount` steps with a default value of 32 steps and a user chosen value via the `--steps` argument. See 3.5 for more details.

7. The result is saved to an output xml file to be viewed with `RRV-visualize`.

3.5 Radiosity Computation

The radiosity computation stage is broken down as follows.

1. The `compute` function is called on a `Scene`- initialized `RadiosityRenderer` object. This function loops `stepCount_` times calling the `computeStep` function on the instance of `RadiosityRenderer` that `compute` was called on. `stepCount_` is set to a default value of 32 unless the argument `--steps` was provided.

2. Inside `computeStep` a loop runs over all `patchCount_` number of `Triangle` objects and sets the previous radiosity to the the current radiosity value. Then for each `Triangle` the current step radiosity value is set following (2.3):
`radiosity` = `reflectivity` · `totalRadiosity(currentPatch_)` + `emission` , where `currentPatch` = 0, 1, ..., `patchCount_`-1 .

3. The `totalRadiosity` method is the routine that computes the summation component of (2.3). The `totalRadiosity` method is called on a `PatchCache` object. The method checks for stored form factor data in the form of a `PatchCacheLine` for the current patch. If it does not exist the `PatchCacheLine` is computed with the `FormFactorEngine` `fillCacheLine` method. The summation in (2.3) is computed by calling the `PatchCacheLine` `totalRadiosity` method on the current `PatchCacheLine` object `cacheLine` along with the current patch index.

   (a) The `PatchCache` class contains a reference to all the `Triangle` structures along with a `std::vector` to hold `PatchCacheLine` instances.

   (b) The `PatchCacheLine` class is a `std::vector` of `std::pair` objects that hold a reference to a `Triangle`'s (previous) radiosity field along with a form factor. This class caches data for quicker computation of the summation component of (2.3) since the form factors only need to be computed once.

(c) The `FormFactorEngine` object `ffe_` is created by the `RadiosityRenderer` constructor and contains a reference to the `std::vector` holding all `Triangle` structures.

4. If the `PatchCacheLine` does not exist for this patch, the `PatchCacheLine` is computed by calling `fillCacheLine` on the `PatchCache` instance of the `FormFactorEngine` `ffe_`. The `fillCacheLine` method renders the environment based on the perspective of the current patch. A container, `map`, is created from a call inside `fillCacheLine` to the `getFF` method. The map is traversed and used to populate the `PatchCacheLine`'s `std::vector`.

   (a) `FormFactorEngine`is a class that uses OpenGL/X11 to render and display patch-perspective views using the hemicube method. Given a patch the patch's center and normal are used to draw the patch's perspective in 5 views.

   A perspective is picked and the transform matrix is created with the aforementioned point and vector with the help of OpenGL functions `gluPerspective` and `gluLookAt`. The container of `Triangle` objects is traversed and each is drawn to the current perspective's sub screen following a transformation and a depth check. The next perspective is chosen and the process is repeated.

5. The `getFF` function reads the on-screen color data to a buffer and maps the form factor coefficients onto this buffer. The buffer is then traversed and the `map` is populated. Pixels of the on-screen data is composed of index-encoded colors. This means that during the draw stage when `Triangle i` is drawn the (24 bit) color is chosen to be `0x00ffffff & i` such that the red component is `0xff & i` the green component is `(i>>8) & 0xff` and the blue component is `(i>>16) & 0xff`. This nontraditional color encoding allows a patch's reflection-component contributors to be computed once for faster rendering.

3.6 Scene Pre-processing

The computation can be conceptually split into two main activities: form factor computation, and radiosity computation. The form-factor computation relies on OpenGL functions to compute transformation matrices. The computation also renders patch-viewpoint images to a X11 window for each side of the hemicube. This process is not only computationally expensive, but it also depends on an OpenGL context, requiring a window to be opened. The form factors were theoretically persistent throughout the execution scope, but the method of storage required an unnecessary degree of spatial complexity that was too high. Because of this, the entire form factor space could not (in reality) be stored in memory for larger scenes. In order to decrease the spatial complexity and facilitate faster rendering through preservation of form factor values with future work-distribution in mind, the methods of both form-factor and `Triangle` structure storage were re-implemented with the use of dynamically allocated arrays of type `float` and type `Triangle` respectively.

With this new storage mechanism for the form factor values, a new method was added to the `FormFactorMatrix` class and this method computes and populates a row-major matrix of form factors. At this point it was realized that the OpenGL/X11 requirements for form factor

computation was inhibitive for server-side rendering as a server-to-client communication was required for each patch-view render and this became a major bottle-neck causing execution time to increase beyond that on a local machine of much lower computational power. The OpenGL/X11 paradigm is also not scalable in a desirable way as parallelization would suggest multiple rendering environments to bring about a decrease in the matrix computation time but since each of these environments would require a separate OpenGL context and hence another window which ultimately leads to decreased performance.

Since the form factors of given scene does not change under varied lighting or reflectivity properties, and are computed once when a scene description is given, we decide to put the focus of our project on the radiosity computation. As a result, the form factors of all the test scenes are computed off-line and stored in files. This allows us to remove the requirement for OpenGL contexts while still allowing for dynamic lighting. Through this process a new data structure, `SceneStructure`, was created to handle loading into and saving of the `Triangle` array, loading and saving of the form factor matrix, and a wrapper for radiosity computation.

Two separate programs are generated by the definition `FF_FROM_SOURCE` supplied with the `-D` compiler flag. If the definition is provided the resulting executable will generate the form factor matrix and save it as a binary file with the name `.ff_mat` appended to the input file's name. Without the definition a form factor matrix with the above naming scheme is assumed to exist and this file is loaded and the radiosity of the scene is computed through a call to `quick_compute` followed by a saving of the resultant scene radiosity to an .xml file (named `output.xml` by default) to be visualized with `RRV-visualize`.

3.7 Iterative Methods for Radiosity Computation

The radiosity equation can be setup as in (2.6) so the computation of a solution involves solving a linear system. We first consider the original method of solving (2.6) which was a fixed-iteration count Jacobi Method. This Jacobi Method was implemented by a nested `for` loop structure where the outer loop ran for a fixed number of iterations and the inner loop computed the current iterator's radiosity element-by-element following (2.3).

Given the radiosity linear system (2.6), the linear system can be defined as (3.1):

$$A\,b = e \tag{3.1}$$

we pick $F = I - A$ or $A = I - F$, so (3.1) becomes

$$(I - F)\,b = e \tag{3.2}$$

and then iterate

$$b^{(k+1)} = e + F\,b^{(k)} \tag{3.3}$$

for $k = 1, 2, ..., M$ where M is fixed. We decided to switch from element-by-element updating to a series of function calls to matrix-vector and *axpby* methods to take greater advantage of automatic loop-vectorization. We created a class `linsolve`, where all linear algebra functions would be stored. Elements in the vectors e and b for the original AOS implementation are tuples of color components (r, g, b) so use of efficient methods such as those in BLAS were initially out of the question. Since the original Jacobi implementation ran for a fixed

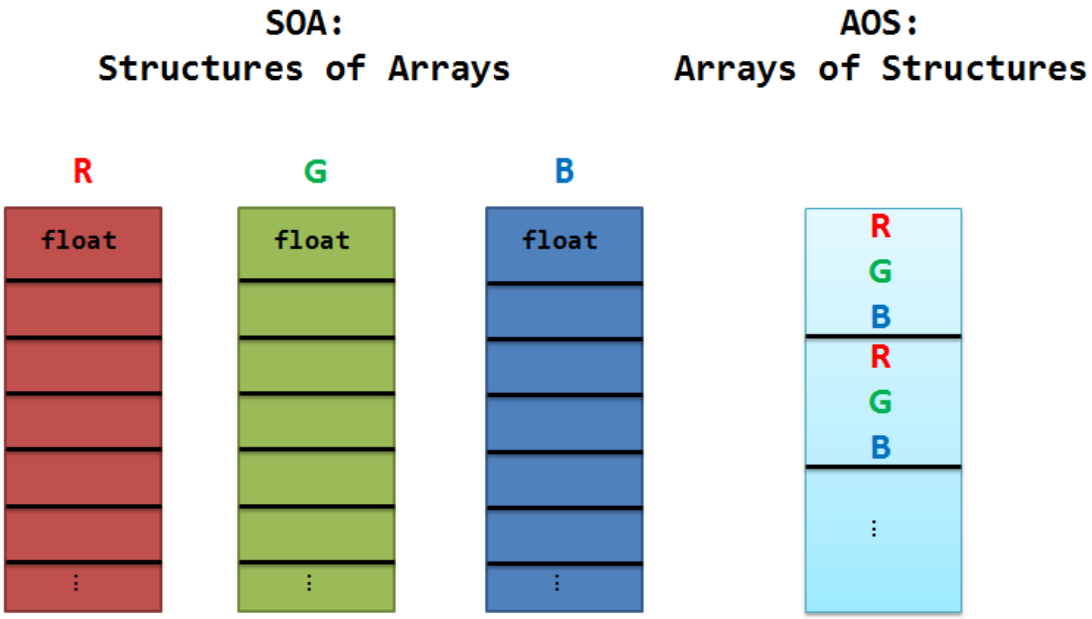

Figure 3.2: Example of Structure of Arrays vs. Arrays of Structures.

number of iterations, a residual calculation is performed so solutions could be quantitatively analyzed. This also cut down on unnecessary iterations for an acceptable solution being reached prior to the specified iteration constant.

The Jacobi method can be slow to converge for complex systems so another iterative method, BiCG-STAB (Biconjugate Gradient Stabilized), was added with the aim of faster convergence and runtime. The BiCG-STAB method does take twice as much work per iteration because of the need for two matrix-vector products per iteration but for complex systems the convergence iteration count for BiCG-STAB should be much less than half the iterations needed for Jacobi, where only one matrix-vector product is needed, resulting in shorter runtimes.

3.8 Distributive Computing Methods

Distributive Computing is the second technique that we used in order to optimize the radiosity computation. This secondary methodology focuses on separating the computationally intensive portions of the program among several processes to benefit from parallelism. Throughout the optimization process, we used CUDA kernels, OpenMP, and CUDA with cuBLAS. The cuBLAS library is a efficient package designed for solving linear systems on NVIDIA GPUs. Given the plethora of options, we have separated our implementations into two categories: Arrays of Structures (AOS) and Structures of Arrays (SOA). We created a Struct of Arrays implementation as it is a typical technique for performance improvements because all elements in the array are contiguous in memory. CUDA and OpenMP both have implementations for SOA and AOS.

3.8.1 Distributive Computing with CUDA

For distributive computing with CUDA we performed a technique known as GPU off-loading, this entails the computationally expensive applications of the program being executed on the GPU while the results and information are returned to the CPU. In this manner, CPU's are only used for the basic operations such as file reading, kernel calls and printing. All matrix and vector operations, including, but not limited to matrix-vector multiplication, dot products and vector scaling are handled on the GPU. In the cuBLAS implementation we utilized the SOA design because it has native float oriented functions for vector-matrix multiplication and dot products known as `cuBLASSgemv` and `cuBLASSdot` respectively. Whereas, when we arranged a AOS implementation we used our own dot product and matrix-vector multiplication because cuBLAS is only designed to work with arrays of floats not structs.

3.8.2 Distributed Computing with OpenMP

The serial version of both SOA and AOS RRV were reimplemented with OpenMP with the goal of speedup. The form factor coefficient matrix can be large so the shared memory model of OpenMP was advantageous so it did not have to be copied around. We only needed to thread computational expensive code portions so only the linear algebra portions of RRV were modified by adding `pragma` declarations to the matrix-vector product, dot product, *axpby*, and element-wise vector scaling.

3.8.3 Hybrid Distributed Computing

When either CUDA or CPU threading are used exclusively the full power of the compute node is not leveraged. Work distribution was implemented by dividing the Jacobi step update process to both threaded CPU methods and GPU-offloading methods. After each step of Jacobi the device and host are synchronized through device-to-host and host-to-device copies.

4 Results

4.1 MPI Poisson

Non-blocking MPI functions on one node were used to solve the Poisson equation in (2.1). As we expected increasing the number of processes per node (Table 4.1) reduced runtime. Minimal runtime reduction is observed when moving from 4 process per node to 8 processes per node. This lack of increase is probably caused by our version of MPI requesting 8 processes on one socket instead of two sockets each with 4 processes, when 8 processes per node is specified. The reason this could result in minimal performance gains is that each socket only has 4 memory channels and so when 8 processes want to access memory there will be a queue for the memory channels. We see that using 16 processes per node results in the best performance.

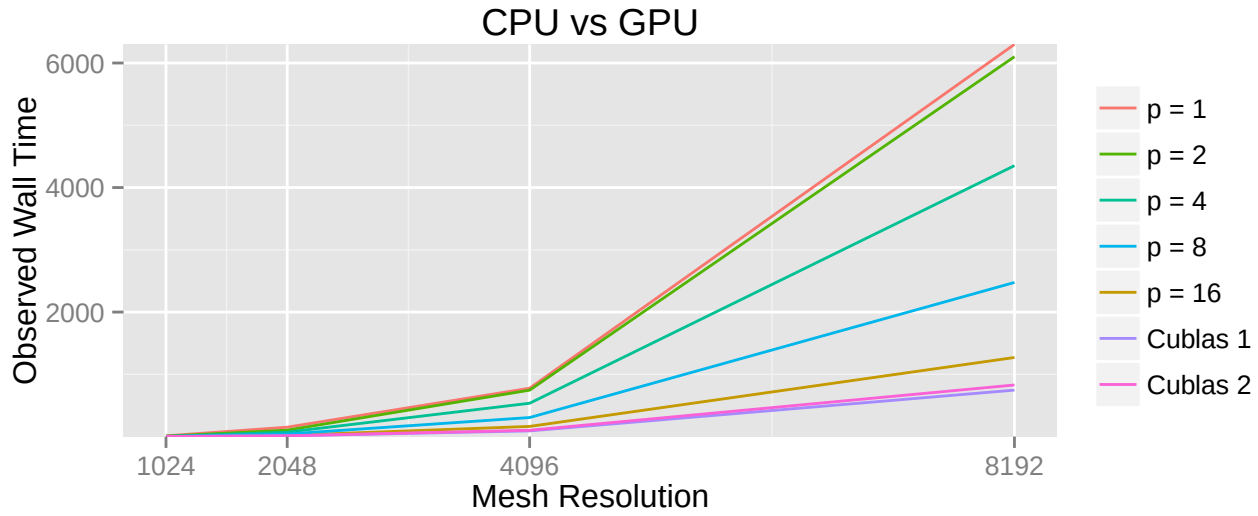

Figure 4.1: Poisson run on one compute node with p processes compare to using GPU with CuBLAS.

Table 4.1: Run times in seconds using p processes and Cuda cuBLAS. The cuBLAS dot product function was used for cuBLAS 1. To compute $y = ax + by$ in cuBLAS 1 we used our own kernel. In cuBLAS 2, however, we used all cuBLAS functions.

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	cuBLAS 1	cuBLAS 2
1024	12.54	7.66	3.96	1.08	0.62	1.836455	2.070277
2048	151.32	103.94	71.86	41.41	19.94	11.897842	13.157421
4096	777.49	745.99	535.98	306.48	163.76	91.892439	102.139394
8196	6299.00	6101.40	4351.30	2475.50	1270.70	747.196718	829.230494

4.2 Single-Node Poisson

Our GPU code clearly outperformed our CPU code. Throughout the tests we were able to see substantial improvements from at a maximum runtime of 6299 seconds to a minimum runtime of 747 seconds with our CPU/GPU codes (Table 4.1). During the implementation of the Poisson equation with CUDA we observed that our own kernels outperform the native cuBLAS kernels. Throughout all cases this is evident; however, for the smallest mesh sizes, GPU code is outperformed by CPU code, probably due to host-to-device communication times. GPU-aided Poisson also show favorable runtime scaling compared to CPU-only Poisson (Figure 4.1).

4.3 Iterative Methods for Radiosity Computation

The radiosity computation is solved with our proposed iterative method. Figure 4.2 presents the results in rendering at different iteration. OpenMP multithreaded versions of the Jacobi and the BiCG-STAB methods were compared with a relative residual tolerance of 10^{-6},

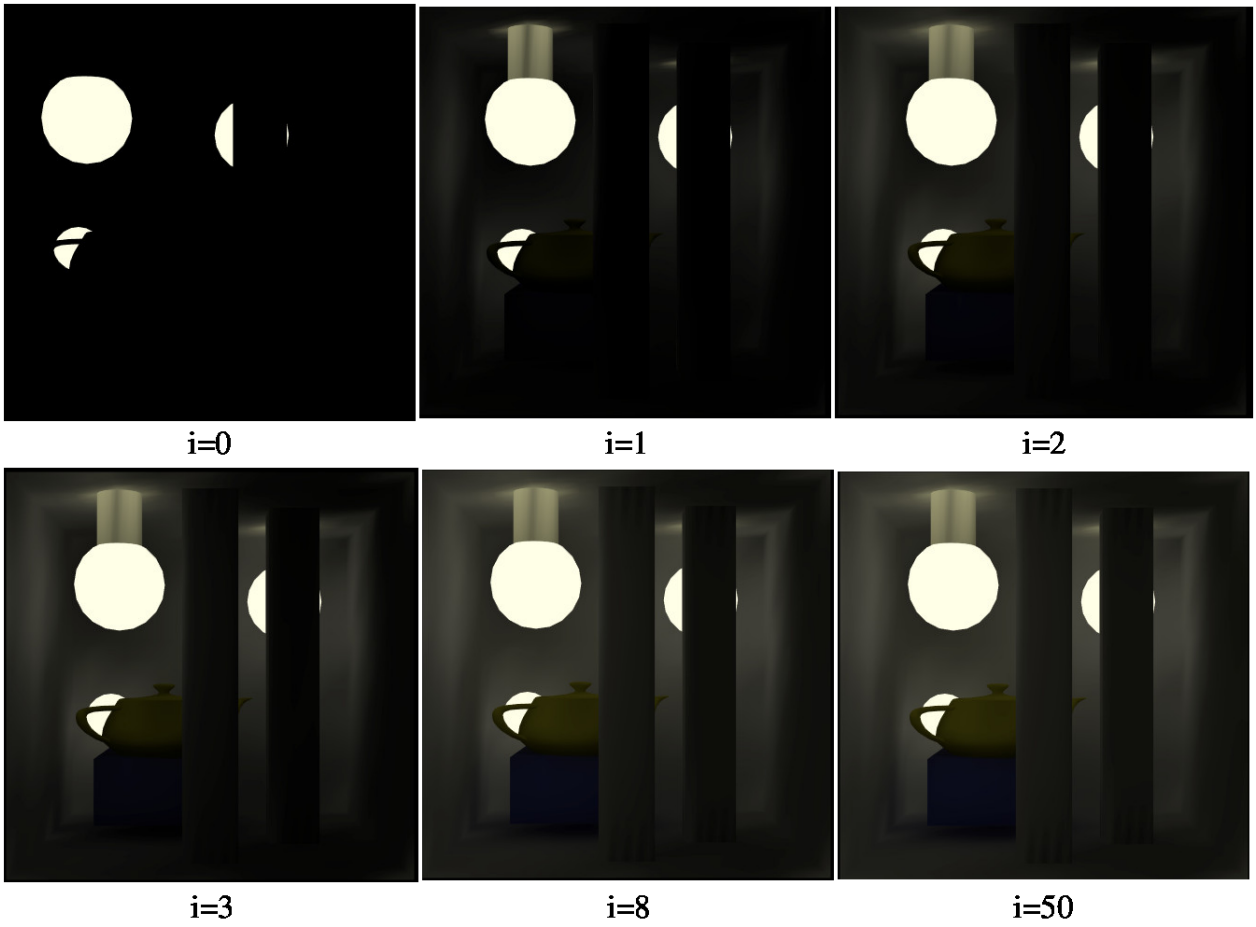

Figure 4.2: Test scene rendering at different iteration.

Table 4.2: Computing Methods Jacobi vs. BiCG-STAB: Time Results for OpenMP with multiple Iterative Methods.

	Patch	Jacobi		BiCG-STAB	
Scene ID	count	iter	runtime	iter	runtime
1	1312	8	0.009	3	0.010
2	3360	28	0.045	16	0.058
3	9680	36	0.410	17	0.435
4	17128	32	0.993	17	1.157

as shown in Table 4.2. BiCG-STAB ran slower in all cases, even with a lower iteration count. The slower runtimes are a result of BiCG-STAB's higher computational complexity per iteration relative to Jacboi. It seems that our scenes were not complex enough to benefit from the reduction in iterations needed for convergence upon swtiching from Jacobi to BiCG-STAB. Therefore, the change in computational methods was ineffective in our project.

Table 4.3: AOS Jacobi Methods: Time Results for All AOS Implementations.

Scene ID	Patch Count	Original	Serial	CUDA	OpenMP
1	1312	0.028	0.031	0.006	0.009
2	3360	0.857	0.677	0.115	0.045
3	9680	10.072	6.973	1.209	0.410
4	17128	27.855	19.394	3.415	0.993

Table 4.4: SOA Jacobi Methods: Time Results for All SOA Implementations.

Scene ID	Patch Count	Original (AOS)	Serial	CUDA	OpenMP
1	1312	0.028	0.040	0.128	0.014
2	3360	0.857	0.913	0.159	0.135
3	9680	10.072	8.821	0.493	0.975
4	17128	27.855	26.809	1.185	2.594

4.4 Distributed Computing for Radiosity Computation and Arrays of Structures

When we used an array of structures both CUDA (no cuBLAS) and OpenMP showed marked improvements on the serial code (Table 4.3). OpenMP show better results than CUDA with the exception of scene 1. We think that communication times from the host to device became inhibitive on larger scene sizes since a device synchronization was required before a transfer could be performed. This removed the performance-gaining asynchronous nature of GPU relative to CPU execution.

4.5 Distributed Computing for Radiosity Computation and Structures of Arrays

CUDA (using cuBLAS libraries) and OpenMP showed marked improvements over our serial code (Table 4.4). CUDA was actually faster than OpenMP for larger problem sizes but performed poorly on small problems. Contrary to general performance enhancement theory, our SOA version for CUDA is faster than our AOS version for CUDA. We speculate that this is due to the lack of a required atomic addition kernel because cuBLAS natively provides matrix-vector and dot product kernels to address these issues. Transfer times with CUDA impede performance due to the fact that all implementations outperform CUDA. Transfer time, in our CUDA scenarios, are greater than the computational time required to reach a solution; however, when larger problem sizes are introduced, the CUDA implementation is clearly justified.

4.6 Hybrid Distributing Computing for Radiosity Computation and Arrays of Structures

A hybrid CPU/GPU Jacobi method was tested by varying the load distribution factor from 0.0 (all CPU, no GPU) to 1.0 (all GPU, no CPU). The execution times (Table 4.5) show

Table 4.5: AOS Hybrid Jacobi Methods: Time Results for Scene 4.

Distribution Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Runtime (s)	1.950	1.458	1.571	1.477	1.695	2.014	2.221	2.502	2.930	3.185	3.515

favorable improvements when using load distribution with respect to the 0.0 and 1.0 distribution cases but for our experiments the device-to-host and host-to-device transfer times could not be overcome to make the hybrid version faster than singularly distributed implementations.

5 Conclusions

We used both computational methods and parallelization techniques to find an optimal solution path for solving for near-real to real time global illumination solutions to the radiosity algorithm. Given the scenes we tested, OpenMP and CUDA both show substantial runtime improvements while the change from the Jacobi method to the BiCG-STAB method actually resulted in increased runtime due to the methods complexity, even while exhibiting faster convergence. It appears that global illumination problems are not in general best suited for mathematical reformulations, though parallelization techniques are quite appropriate and give favorable speedups compared to initial serial code.

6 Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (www.umbc.edu/hpcreu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2014. This program is funded jointly by the National Science Foundation and the National Security Agency (NSF grant no. DMS–1156976), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-authors Oluwapelumi Adenikinju and Joshua Massey were supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Co-authors Jonathan Graf, Xuan Huang, Samuel Khuvis, and Yu Wang were supported during Summer 2014 by UMBC.

References

[1] David Bařina, Kamil Dudka, Jakub Filák, and Lukáš Hefka. RRV — Radiosity Renderer and Visualizer, 2007. http://dudka.cz/rrv/, accessed on August 07, 2014.

[2] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, 1984.

[3] Donald P. Greenberg, Michael F. Cohen, and Kenneth E. Torrance. Radiosity: a method for computing global illumination. *The Visual Computer*, 2:291–297, 1986.

[4] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report HPCF–2014–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[5] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processes: A Hands-on Approach*. Morgan Kaufmann, 2010.

[6] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[7] NVIDIA Developer Zone. Parallel thread execution, 2006. http://docs.nvidia.com/cuda/parallel-thread-execution/graphics/memory-hierarchy.png, accessed on July 24, 2014.