Concurrent Solutions to Linear Systems using Hybrid CPU/GPU Nodes

Oluwapelumi Adenikinju[1], Julian Gilyard[2], Joshua Massey[1], Thomas Stitt[3]

[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

Abstract

We investigate the parallel solutions to linear systems with the application focus as 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 basic Jacobi method with a fixed iteration count 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 for complex problems. It is found that for the test scenes used, the problem complexity was not great enough to take advantage of mathematical reformulation through BiCG-STAB. Single-node parallelization techniques are implemented through OpenMP-based multi-threading, GPU-offloading using CUDA, and hybrid multi-threading/GPU offloading. It is seen that in general OpenMP is optimal by requiring no expensive memory transfers. Finally, we investigate two storage schemes 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 various strategies to solve problems that contain multiple linear systems. The global illumination computer graphics problem is chosen as the case study with the radiosity method used as the solution approach. Illumination solutions are necessary in the rendering timeline for applications in video games, animated movies, and CGI. A global illumination approach was picked instead of a local illumination solution to allow light to reflect prior to hitting the eyes of the viewer, giving a more realistic description of the environment. Illumination solutions can be decomposed into tuples of red, green, and blue intensities for full color visualization.

The basic Jacobi method with a fixed number of iterations is first used following the solver provided [1]. We propose to use a modern iterative linear solver as an 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 (application program interface), and exploit access to massively parallel

hardware through GPU-offloading with CUDA, in which data are transfered ("offloaded") to the GPU for processing. The combination of GPU-offloading and CPU-threading is explored through a hybrid CPU/GPU compute implementation.

Performance metrics are executed and 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.

The outline of the report is the following. Section 2 is devoted to explanation of background and problem statement. We will first introduce the desire for currently solving linear systems and then discuss our case study, the radiosity method. Section 3 will explain the computational methods and parallel techniques used. Results using array of structures and structure of arrays will be discussed respectively in Section 4. Section 5 summarizes our conclusions and motivates future work.

2 Background and Problem Statements

The global illumination problem in computer graphics involves solving for the steady-state light distribution in a given graphical environment created previously in design software. In other words, we are solving for the light intensity given off of each patch in a graphical environment definition. Various methods exist for solving illumination problems, but the radiosity method gives better results by allowing light to reflect off surfaces prior to hitting the viewers eye. This makes the radiosity solution a global illumination solution.

The radiosity method (Cindy Goral et al. [2]) is given by one vector equation for each patch (collection of pixels) of each color, indicated by $\nu = 1, 2, 3$ for red, green, blue, respectively:

$$B_i^{(\nu)} = E_i^{(\nu)} + \rho_i^{(\nu)} \sum_{j=1}^{N} B_j^{(\nu)} F_{i,j} \qquad \text{for } i = 1, 2, \ldots, N, \tag{2.1}$$

where

- $B_i^{(\nu)}$ (radiosity) is the total energy leaving the surface (radiosity) of the ith patch (energy/unit time/unit area) of the ν^{th} color,
- $E_i^{(\nu)}$ (emission rate) is the inherent rate of energy leaving the ith patch (energy/unit time/unit area) of the ν^{th} color (nonzero iff this patch is a light source),
- $\rho_i^{(\nu)}$ (reflectivity) is the reflectivity of the ith patch (unitless) of the ν^{th} color. 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.

Here, the values of $\rho_i^{(\nu)}$ are frequency dependent and general environmental illumination is not monochromatic hence we see the need for multiple solutions to (2.1) over all patches for each color needed.

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

$$\begin{bmatrix} B_1^{(\nu)} \\ B_2^{(\nu)} \\ \vdots \\ B_N^{(\nu)} \end{bmatrix} = \begin{bmatrix} E_1^{(\nu)} \\ E_2^{(\nu)} \\ \vdots \\ E_N^{(\nu)} \end{bmatrix} + \begin{bmatrix} 0 & \rho_1^{(\nu)} F_{1,2} & \cdots & \rho_1^{(\nu)} F_{1,N} \\ \rho_2^{(\nu)} F_{2,1} & 0 & \cdots & \rho_2^{(\nu)} F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_N^{(\nu)} F_{N,1} & \rho_N^{(\nu)} F_{N,2} & \cdots & 0 \end{bmatrix} \begin{bmatrix} B_1^{(\nu)} \\ B_2^{(\nu)} \\ \vdots \\ B_N^{(\nu)} \end{bmatrix}, \tag{2.2}$$

which can be written as a linear system

$$A^{(\nu)}\, b^{(\nu)} = e^{(\nu)} \tag{2.3}$$

with $A^{(\nu)} = I - G^{(\nu)}$ and

$$G^{(\nu)} = \begin{bmatrix} 0 & \rho_1^{(\nu)} F_{1,2} & \cdots & \rho_1^{(\nu)} F_{1,N} \\ \rho_2^{(\nu)} F_{2,1} & 0 & \cdots & \rho_2^{(\nu)} F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_N^{(\nu)} F_{N,1} & \rho_N^{(\nu)} F_{N,2} & \cdots & 0 \end{bmatrix}, \quad b^{(\nu)} = \begin{bmatrix} B_1^{(\nu)} \\ B_2^{(\nu)} \\ \vdots \\ B_N^{(\nu)} \end{bmatrix}, \quad e^{(\nu)} = \begin{bmatrix} E_1^{(\nu)} \\ E_2^{(\nu)} \\ \vdots \\ E_N^{(\nu)} \end{bmatrix}.$$

An open-source program called Radiosity Renderer and Visualizer (RRV) [1] was used as a base solver. RRV is a serial C++ and OpenGL global illumination solver that takes xml files defining a scene and generates an xml file describing the lit scene. RRV along with the scene files used are available at http://dudka.cz/rrv. Scenes were sized from low thousands to tens of thousands of patches and were limited by the GPUs memory as we did not consider problems large enough to require paging in and out data.

All code was written in C/C++ with NVIDIA's and OpenMP's language extensions. The compilers used were Intel's C++ compiler icpc and NVIDIA's CUDA compiler nvcc.

3 Methodology and Parallel Implementation

3.1 Hybrid CPU/GPU Computing Methods

Hybrid CPU/GPU computing is one method of realizing performance gains independent of the iterative method used. With hybrid CPU/GPU computing, we focus on separating the computationally intensive portions of the program among several workers. From the standpoint of solving multiple linear systems, hybrid CPU/GPU computing can achieve significant speedup by using high-level OpenMP and CUDA library functions, therefore giving a relatively simple implementation. The compute nodes available had multicore processors and GPUs so both shared-memory CPU threading GPU-offloading were considered.

Here we pause and discuss considering the layout of data in memory and its affect on the methods used. Two schemes can be used to map multi-dimensional data into memory called arrays of structures (AOS) and structures of arrays (SOA). A visual representation for our system can be seen in Figure 3.1. AOS means that each array entry is a tuple containing an entry for each piece of information, which is the red, green, and blue color intensity in the scope of this paper. Mathematically, the unknown vector b in the linear system $A\, b = e$

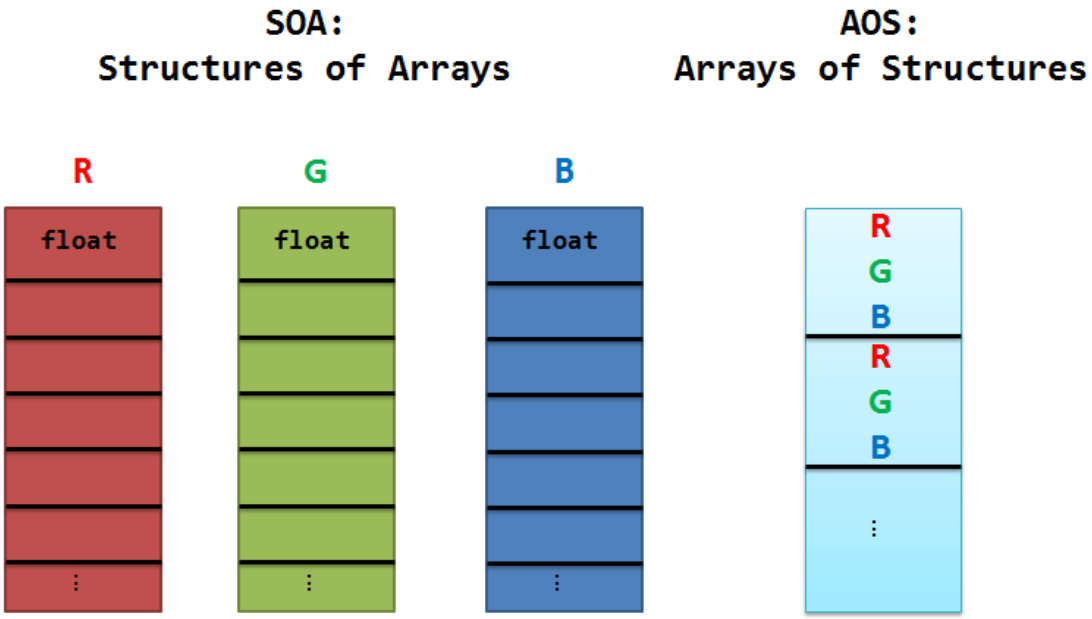

Figure 3.1: Example of structure of arrays vs. array of structures.

lists the entries of the $B^{(\nu)}$, $\nu = 1, 2, 3$, in interleaved ordering. The right-hand side e is then also set up with interleaved ordering, and the system matrix A in $A\,b = e$ contains the elements of $A^{(\nu)}$, $\nu = 1, 2, 3$, interleaved in both dimensions. By contrast, SOA means that each piece of information is stored in its own array and the arrays are stored in a single structure. In our case, these are the three arrays: red, green, and blue. Mathematically, this means that the unknowns $B^{(\nu)}$, $\nu = 1, 2, 3$, are stacked on top of each other to form the vector of unknowns b. Analogously, the right-hand side e is obtained, and the system matrix A in $A\,b = e$ is a 3×3 block matrix, whose diagonal blocks are the matrices $A^{(\nu)}$, $\nu = 1, 2, 3$. In either case, we can consider the data as one object, either an array or a structure, for easy passing between methods. When a structure of arrays is used for the data representation the problem is simply a set a one-dimensional linear systems so linear solver packages like BLAS and cuBLAS (BLAS CUDA kernels) can be used.

3.1.1 GPU Computing with CUDA

With access to NVIDIA GPUs we used offloading to leverage the massively parallel architecture of the GPUs with the CUDA architecture [5]. GPUs have their own memory and clock and offloading refers to transferring relevant memory to the GPUs memory and running a kernel (function) on the data. GPUs are designed for single instruction multiple data (SIMD) routines because each core follows the same program counter (PC) in its warp (collection of cores). In this manner, the CPU code can be used for the basic operations such as file manipulation, user I/O, and conditional heavy subroutines while the GPU does simple computationally expensive tasks like the linear algebra routines used in solving linear systems. All matrix and vector operations, including, but not limited to, matrix-vector multiplications, vector products, and vector scalings are handled on the GPU. The fundamental

challenge of GPU offloading is that the computational work performed by the GPU needs to be so fast that it overcomes the delay caused by the memory transfer.

The CPU handles kernel calls by executing a series of functions for the specific linear solver and afterwards requests the current solution's residual back with a `cudaMemcpy` call. This residual is compared to the tolerance, constituting one iteration of the iterative method. Prior to the solving procedure, all device memory is allocated and the matrix and initial solution guess are copied to the device. Kernels, the functions on a GPU, are launched in a so-called grid of thread blocks. We use 1D grid and block layouts. Each block has 512 threads, and an appropriate number of blocks per grid were calculated based on the patch count with one thread created for each patch. For example, a scene with 2048 patches would be placed on one 1D grid where the grid would have four blocks, each with 512 threads. Additionally, the grid dimension was calculated to be the ceiling of the patch count divided by the block size so that there would indeed be one thread per patch.

In the case of SOA memory, the cuBLAS library is utilized since the arrays map easily onto the matrix-vector product (`cuBLASSgemv`) and vector-vector product (`cuBLASSdot`) routines [4].

When we used AOS memory, we used our own vector-vector product and matrix-vector product kernels because cuBLAS is incompatible with the arrays of structures format.

3.1.2 CPU Computing with OpenMP

The serial version of both the AOS and SOA solvers were reimplemented with OpenMP [6] to take advantage of available cores not utilized in serial execution. The form factor coefficient matrix is large with respect to scene size ($O(n^2)$) so the shared memory model of threading with OpenMP was advantageous to avoid communication. We needed only to distribute the computationally expensive routines, therefore only the linear algebra portions of the solver were updated, including the matrix-vector product, dot product, *axpby*, and element-wise vector scaling methods, with the appropriate `#pragma parallel for` and `reduction` declarations.

3.1.3 Hybrid CPU/GPU Computing

When we use a solely GPU-offloading or CPU-threading solver, the full power of the hybrid CPU/GPU compute node is not leveraged. In order to utilize the entire node, we implemented a hybrid solver by dividing the (in our case) faster Jacobi method to use both OpenMP and CUDA routines. A distribution factor (between 0 and 1) is used to control the splitting of the workload between the GPU and CPU where 0 is a completely CPU routine and 1 is a complete GPU routine. After each step, there are two vectors whose lengths sum to N. This requires one synchronization between the memory of the CPU and GPU per iteration.

3.2 Iterative Methods

Another method to realize performance gains is to consider the mathematical method used to solve the linear systems. Matrix inversion is an $O(n^3)$ operation and hence is not efficient

when iterative methods requiring $O(n^2)$ operations per step can achieve a solution in much fewer than n steps. This is true for our case study of the radiosity method for global illumination solutions.

We first consider the original method of solving the combined linear system $A\,b = e$, with b either in AOS or SOA arrangement, with a Jacobi iterative method with a fixed-iteration count of M iterations, as implemented in the original code [1]. We introduce $G = I - A$ or $A = I - G$, so that $A\,b = e$ becomes $(I - G)\,b = e$ or

$$b = e + G\,b. \tag{3.1}$$

The fixed-iteration Jacobi method then iterates

$$b^{(k+1)} = e + G\,b^{(k)} \tag{3.2}$$

for $k = 1, 2, \ldots, M - 1$, 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. Elements in the vectors e and b for the original AOS implementation are tuples of color components (r, g, b) so use of existing functions such as those in BLAS were initially out of the question.

Since the original Jacobi implementation ran for a fixed 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 is the simplest of the so-called basic iterative methods [7, Ch. 4] and can be slow to converge for complex systems. So, another iterative method, the modern BiCG-STAB (Biconjugate Gradient Stabilized) [7, Ch. 7], 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 we hope that the iteration count for BiCG-STAB would be less than half the iterations needed for Jacobi, where only one matrix-vector product is needed, resulting in shorter runtimes.

4 Results and Discussion

4.1 Iterative Methods

The radiosity computation is solved with our proposed iterative methods. Four scenes of varying complexity and patch count are used. Figure 4.1 presents the results in rendering at different iteration counts using the Jacobi method for Scene 3. The initial image shows the initial guess, which is only the light from light sources. Algorithmically, the iterations in the radiosity calculation with the Jacobi method follow the light, as it bounces around the scene, so that the final converged image shows the light, as it is attached to all objects in the scene.

OpenMP multi-threaded versions of the Jacobi and the BiCG-STAB methods were compared with a relative residual tolerance of 10^{-6}, as shown in Table 4.1. BiCG-STAB ran

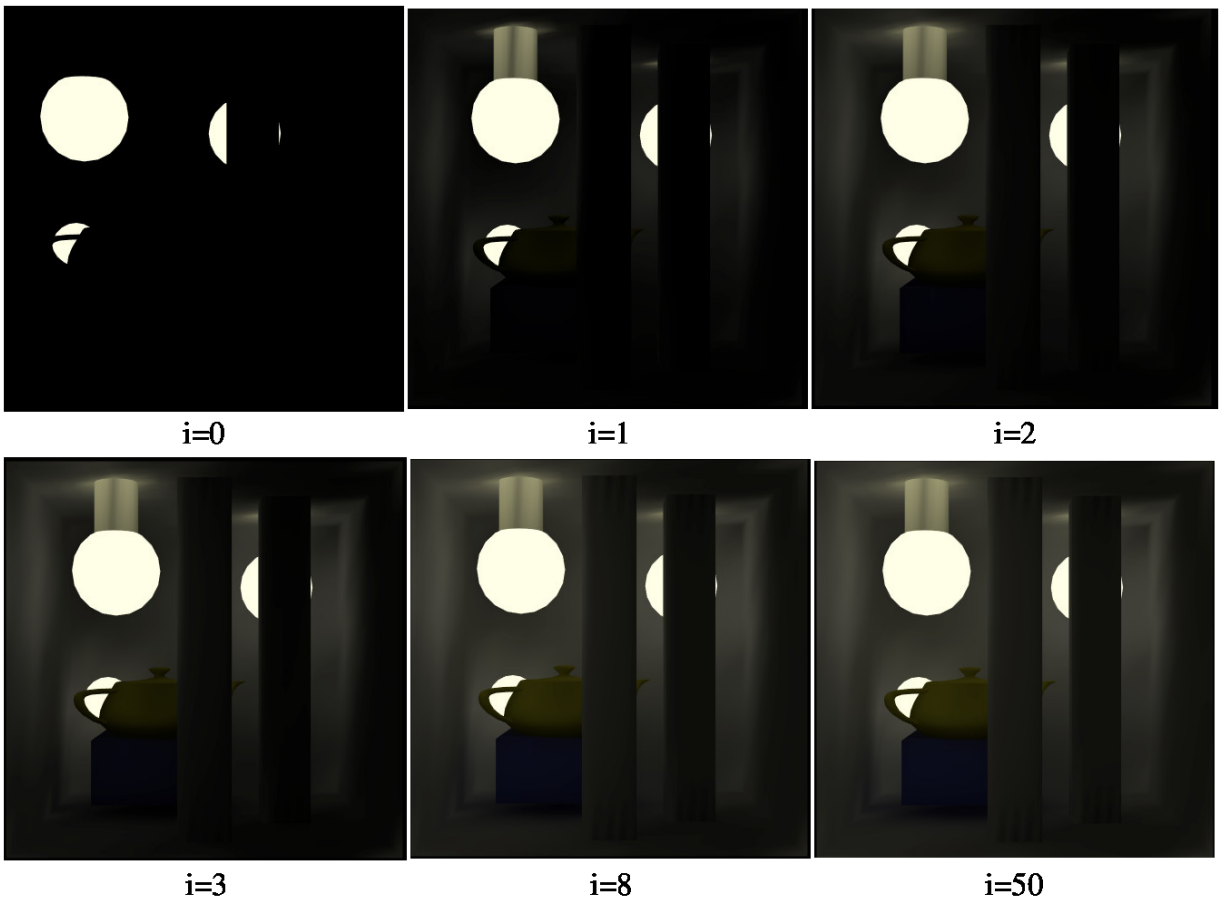

Figure 4.1: Test scene rendering at different iterations for Scene 3.

Table 4.1: Iterative methods Jacobi vs. BiCG-STAB: timing results for OpenMP with runtimes in seconds.

	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

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 Jacobi. It seems that our scenes were not complex enough to benefit from the reduction in iterations needed for convergence upon switching from Jacobi to BiCG-STAB. Therefore, the change in computational methods was ineffective in our project.

Table 4.2: AOS Jacobi methods: timing results in seconds 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 4.3: SOA Jacobi methods: timing results in seconds 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

4.2 CPU and GPU Computing with Arrays of Structures

When we used an array of structures for the memory layout both CUDA (no cuBLAS) and OpenMP showed marked improvements over the serial solver. Table 4.2 details our results and we see that OpenMP showed better runtimes than CUDA with the exception of Scene 1. Our custom matrix-vector product CUDA kernal suffered for larger scene sizes because of uneven data distribution across GPU cores. Because of this, reduced parallelism was present with larger scenes, but improvement could be realized with better load balancing through an improved reduction routine as shown by Harris [3].

4.3 CPU and GPU Computing with Structures of Arrays

CUDA (using cuBLAS libraries) and OpenMP showed marked improvements over our serial code as seen in Table 4.3. CUDA was actually faster than OpenMP for larger problem sizes but performed poorly on small problems. Surprisingly, CUDA does not trend like that in the AOS case, showing better results for the larger two scenes compared with OpenMP. Although this is the case, overall runtimes are still slower than those with the AOS memory layout and OpenMP.

4.4 Hybrid CPU/GPU Computing

A hybrid CPU/GPU AOS Jacobi method was tested by varying the load distribution factor from 0 (all CPU, no GPU) to 1 (all GPU, no CPU) by steps of 0.1. The execution times, shown in Table 4.4, show favorable improvements when using load distribution factors between 0.2 and 0.4, 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 a solely CPU or GPU implementations. This can be seen plainly in the case of a 0 distribution factor since here we are only opening connections and transferring data. Quantitatively we compare this on Scene 4 from our non-hybrid AOS OpenMP results in Table 4.2, which takes 0.993 seconds

Table 4.4: AOS hybrid Jacobi methods: timing 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

in a pure OpenMP case, but 1.950 seconds in OpenMP with communication with the GPU.

5 Conclusions

We used computational reformulations as well as hybrid CPU/GPU computing techniques in the search of an optimal solution for a group of systems. Our case study was the radiosity method for solving the global illumination problem consisting of three systems. 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 method's complexity, even while using fewer iterations. 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 the initial serial code.

An array of structures memory layout was found to be faster, but one should be careful if not using the entire structure for a solution as it will take up unnecessary space in cache. OpenMP was found to be faster than CUDA with AOS, but following the trend in the SOA case we believe that with a better reduction routine the CUDA AOS case could be made faster following results shown by Harris [3].

A hybrid CPU/GPU AOS implementation was tested which involved distributing portions of the matrix to the CPU and GPU. Communication was too much of a bottleneck to achieve a solution faster than the OpenMP case.

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. The project problem was proposed by Yu Wang and Dr. Marc Olano from the Department of Computer Science and Electrical Engineering at UMBC. Our team worked on the problem with the support of graduate assistants Jonathan

Graf, Samuel Khuvis, and Xuan Huang, and faculty mentor Dr. Matthias K. Gobbert from the Department of Mathematics and Statistics at UMBC.

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 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.

References

[1] D. Bařina, K. Dudka, J. Filák, and L. Hefka, *RRV — Radiosity Renderer and Visualizer*, 2007. Documentation of and orginal source for the radiosity solver, http://dudka.cz/rrv/, accessed on January 07, 2015.

[2] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, *Modeling the interaction of light between diffuse surfaces*, Computer Graphics, 18 (1984), pp. 213–222.

[3] M. Harris, *Optimizing parallel reduction in CUDA*, 2007. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, accessed on January 07, 2015.

[4] NVIDIA, *cuBLAS: CUDA toolkit documentation*, 2007. http://docs.nvidia.com/cuda/cublas/, accessed on January 07, 2015.

[5] *NVIDIA Developer Zone, Parallel Thread Execution*, 2006. http://docs.nvidia.com/cuda/parallel-thread-execution/graphics/memory-hierarchy.png, accessed on January 07, 2015.

[6] *OpenMP Architecture Review Board, OpenMP Application Program Interface*, 2008. http://www.openmp.org/mp-documents/spec30.pdf, accessed on January 07, 2015.

[7] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, second ed., 2003.