Global Illumination simulation on multi core computers*

B.Kh. Barladian, L.Z. Shapiro, E.Yu. Denisov, A.G. Voloboy
bbarladian@gmail.com, eed@gin.keldysh.ru, pls@gin.keldysh.ru, voloboy@gin.keldysh.ru
Keldysh Institute of Applied Mathematics RAS, Moscow

Global Illumination (GI) simulation is time consuming. Taking into account that modern computers contain tens of cores the simulation algorithms should be effectively distributed. The paper describes GI simulation algorithm specially tuned for spectral simulation on the multi core computers. Results of simulation are stored in the Illumination Map form. The algorithm dynamically creates additional threads for the critical branches of the computational process. This leads to balanced loading of the threads and finally ensures a full load of physical and virtual cores.

Keywords: *realistic rendering, global illumination, Illumination Map, hyper spectral simulation, multi-threaded simulation.*

1. Introduction

All modern realistic visualization systems utilize global illumination (GI) for generating of highly realistic images. In our system we calculate the GI by forward Monte-Carlo ray tracing [1]. And resultant illuminance distribution is stored in the so called Illumination Maps (i-maps) in view-independent way [2]. GI simulation is the most costly procedure and requires a significant amount of calculations for realistic rendering as well as for optic simulation tasks. Modern computers offer today two options for calculation acceleration: general purpose GPU usage and multi-core, multi-threaded processing. Both approaches have own advantages and disadvantages. Using GPU requires, as a rule, essential rewriting of existing algorithms [3]. Effective implementation of a complex optical model of surface and medium on GPU is a difficult task. Especially it concerns using large 4-dimensional tabulated a Bidirectional Reflection Distribution Function which describes arbitrary optically complex surfaces, simulation of volume scattering, fluorescence, dispersion phenomena and so on. GI algorithms on GPU typically are used on notebooks with powerful modern graphic cards in game and animation applications. In the same time the realistic rendering and optic simulation systems, where physically accurate simulation with high accuracy is required, often use powerful workstations like HP DL380z Generation9 (Gen9) Virtual Workstation [4]. Basing on Intel® Xeon® Processor E5-2600 v4 [5] these workstations can used up to 22 cores and 44 threads per socket, totally up to 88 threads for calculations. Our main GI algorithm – forward Monte Carlo ray tracing – can be easily implemented for multi-threaded calculations. In this case each thread performs ray tracing independently. A scene description (geometry, acceleration structures, light sources, optical properties of surfaces and media and so on) is not changed during simulation. Therefore the shared memory is most suitable for the scene data storing.

However there is a problem with storing of the simulation results in the form of Illumination Map. Our system uses rays (photons) with unit energy. The correct result of the simulation is achieved by appropriate selection of probabilities for the light events such as creation a light ray, its reflection, refraction or absorption on surfaces, absorption and scattering of light in the environment [6]. At the moment of the ray intersection with the triangle (i.e. with the element of Illumination Map) the energy stored in the triangle vertices is modified in accordance with the energy delivered to the surface by the ray. The only Illumination Map for all threads is stored in the shared memory because the amount of memory necessary for its storing is big enough. If we calculate i-maps in the RGB mode then for each triangle vertex and for the each surface side the four values must be stored in double precision format. The three values of them are needed to store the RGB values and the fourth one is needed to estimate the accuracy achieved by the Monte Carlo ray tracing simulation. Double precision format is required to avoid the loss of precision in the energy accumulation when billions of rays are used in simulation. In the spectral simulation the number of color values has to be equal to the number of wavelengths used in simulation. Thus to store the Illumination Map for a scene with 1000000 vertices it is necessary 64Mb for simulation in the RGB mode and 640 Mb for spectral simulation with 40 wavelengths.

In multi-threaded Illumination Map calculation each thread performs Monte Carlo ray tracing for a limited number of rays and accumulates a portion of ray-triangle "intersection objects". The object describing intersection contains information about the triangle and geometric object indices, the barycentric coordinates of the ray-triangle intersection point and the ray color. These portions (let's define them IOP – Intersection Objects Portion) are independently

The work was supported by RFBR, Grants No. 15-01-01147, 16-01-00552 and Integra Inc. company. Работа опубликована по гранту РФФИ №16-07-20482

calculated in each thread. Then they are stored in a stack from which they were extracted and processed by main thread. The IOP processing included:
1. calculation of contribution of ray-surface intersection to the energy stored in the triangle vertices according to the barycentric coordinates of the intersection point;
2. addition of the calculated contribution to the respective energy values accumulated in the triangle vertices.
Since the access to the arrays of energies accumulated in the triangles vertices is made from the one main thread only then there is no need in synchronization with other threads. In this approach synchronization is required only between the threads putting calculated portions into the stack and the main thread which removes the portions for subsequent processing (Fig. 1).

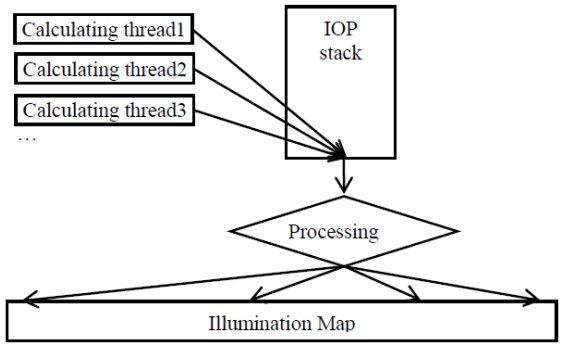


Рис. 1: Classic multithreaded computing algorithm.

Synchronization in these procedures is provided by the common critical section. As the procedures to put the portion in stack and to get it are a simple pointer copying, they are very fast and thus conflicts and performance losses in these procedures do not arise. We call the algorithm of the multithreaded Illumination Map computing represented on Fig. 1 as a classic one. The bottleneck in this approach is the performance of the main thread that is IOP processing. This approach usually works quite effectively on computers with a few (two or four) physical cores but even for eight cores the computer can be not fully loaded. The Task Manager shows performance degradation in this case (Fig. 2).

The evident solution is to increase the number of threads that process calculated IOP. However direct using of several threads for IOP processing will require synchronization of access to the array of energies accumulated in the triangles vertices. Using synchronization here will decrease the performance significantly. To solve the problem the new algorithm was suggested.

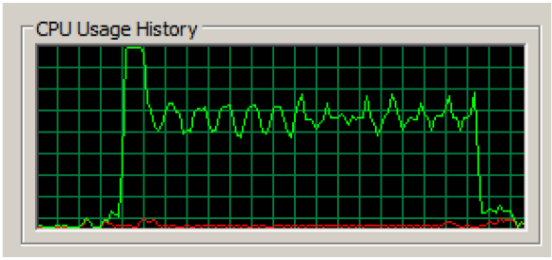


Рис. 2: The loss of performance for classic algorithm.

2. Proposed algorithm

The scheme of the proposed algorithm is presented on the Fig. 3.

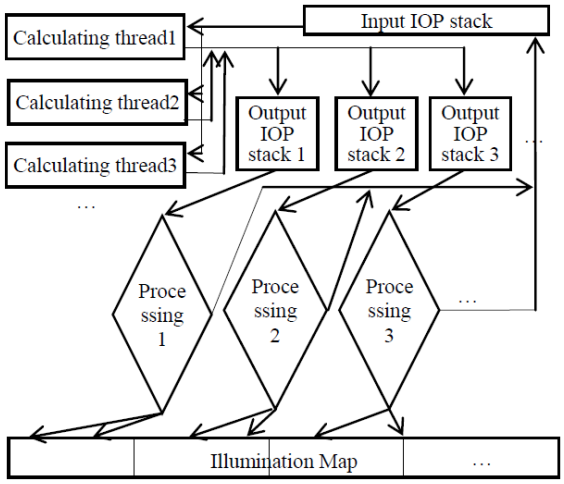
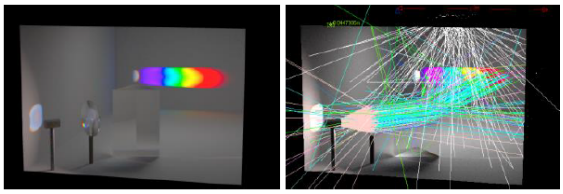


Рис. 3: New multithreaded computing algorithm.

We subdivide the whole set of the scene vertices on the several groups. It can be said that we subdivide Illumination Map. Now the energy addition to the energy accumulated in the vertices can be done for each group by separate thread independently and without critical section. The first part of the IOP processing (calculation of the contribution of ray surface intersection) is moved now to the computing threads. This reduces the loading on the threads which process the calculated portions. The calculated vertex energy contributions we divided into the non-overlapping groups corresponding to the different vertex groups in the i-maps. Since the groups of vertices are not intersected then there is no need for thread synchronization during adding these simulation results to the corresponding vertex groups. The synchronization is needed only for the transfer of new portions of energy additions to the new threads carrying out this addition to the triangle vertices. But as it was mentioned above this is simple pointer copying which is a very fast procedure and thus conflicts and performance losses do not arise here.
Initially the vertices of the triangles can be divided into equal groups by using natural vertex numeration.

To control the work with memory we use some fixed number of IOP to store vertex energy additions. These IOPs are initially created and stored in the input IOP stack and are gotten from it by calculating threads. After filling them by results they are put into the appropriate output stack. Appropriate processing thread gets IOP and after adding the energy to the Illumination Map returns it to the input IOP stack. The suggested approach of the IOP processing is quite effective for most of the practical scenes where photon number for different vertex groups varies not very much.



(a) Ray traced image. (b) Ray paths visualization.

Рис. 4: Scene with nonuniform photon distribution.

However in some cases the full processor loading cannot be reached. Appropriate example is the scene "dispersion" presented on Fig. 4. The reason is essentially nonuniform distribution of photons on the vertex groups. Some threads are forced to process a significantly more photons than others. Finally the IOP processing is slower than its creation by calculating threads. As a result the calculating threads are caused to wait until processing threads return the IOP to the input stack.

This problem is critical in case of a large number of threads when the vertices are divided into a relatively large number of groups. The first evident solution is to define the statistical distribution of photons on triangles vertices and perform the subsequent vertices partitioning into the groups based on these statistics. However this approach requires special preliminary calculation step to get the necessary statistics, rebuilding data structures for optimum partitioning of vertices. Only after that the final calculations will be started. It is unlikely that we can avoid idle processors in such an approach. At the same time our goal is not to obtain an optimal vertices partitioning on the groups but to get the most efficient usage of the computing threads. Nonuniform loading of processing threads or even their stoppage time from time is not critical for the system. The effectiveness of the system is determined, mainly, by a full loading of calculating threads. In most cases their number is chosen equally to the total number of computer cores (physical and virtual).

To solve the problem of nonuniform loading of the threads we elaborated an adaptive algorithm of dynamical partitioning the vertices in the groups without stopping the calculating threads.

3. Dynamical partitioning of vertices

At the first step the vertices are divided into a finite number of groups. The group index for a given vertex is determined by simple dividing the vertex index on the number of vertices in the group. Then the structures required for the calculations are prepared. The main structures are the input IOP stack and an array of the output IOP stacks. The dimension of the array of the output IOP stacks equals to the number of the vertex groups.

Calculating thread gets one portion for each group from the input IOP stack and performs Monte Carlo ray tracing until at least one portion is filled. Then each calculated portion is put into the output IOP stack corresponding to a given vertex group. Each processing thread adds simulation results obtained from the portion into accumulated energy arrays of the respective vertex group. After processing the portion is cleaned and returned to the input IOP stack.

If one of the processing threads processes incoming portions slower than the calculations threads create them then the length of its output IOP stack is increasing and the length of the input IOP stack is respectively decreasing. At some moment the portion number in the input IOP stack becomes insufficient for work of calculating threads. So they have to wait while processing threads provide necessary number of portions.

This problem is detected if length of the input IOP stack becomes below a certain threshold. It is solved by splitting of the problematic vertex group. The group is split into a number of new ones and new processing threads should be created. The group is determined by the length of the output IOP stack which exceeded a predetermined threshold. The simple but effective splitting method was selected: splitting into a fixed number of sub-groups with the same number of vertices of triangles. Let us denote this parameter as subd for further discussions and formulas. Experiments with practical scenes using computers with various numbers of cores showed that in most cases the division into four subgroups provides fast convergence of the algorithm and allows implementing an efficient algorithm for determining the index of group by vertex index.

After splitting the group index cannot be defined by simple division of the vertex index. Firstly we try to use array of group indices for all vertices. However experiments showed that using of this array reduces the tracing speed about 5-7%. This problem is related to the caching of memory in the case of multi-core processors because simple calculations take less time than index-based access to large arrays in the shared memory. So in the dynamic groups partitioning we

developed a recursive algorithm corresponding to the recursive partitioning groups into subgroups. For a description of each group of vertices the following structure is used:

struct VertexLevelSubd { int subd_state; // (0 - not subdivided, 1 - subdivided). int vert_indx; // Index of first vertex in the group. int vert_number; // Number of vertices in given group. int first_thread_indx; // Index of first new subgroup };

At the first step all the vertices are divided into equal groups and initial array of VertexLevelSubd structures is created. The structure fields are initialized as follows: subd_state = 0 – the group is not subdivided; vert_indx and vert_number are set in accordance with their actual values for this group;
first_thread_indx = -1 – it is not defined until the group splitting.
During splitting of the given group into the subd subgroups the subd new processing threads are created. Thread index corresponds to the new subgroup index. Array of the output IOP stacks also is extended by subd elements corresponding to the new vertex groups. The field first_thread_indx is set to the index of the first new thread. The array of VertexLevelSubd structures also is extended by subd elements. The subd_state value of VertexLevelSubd structure is set to “1” for the subdivided group. Calculating threads do not put new portions to the output IOP stack of subdivided group. They will start to fill the output IOP stacks for new created groups. It should be noted that the field subd_state is used simultaneously by calculating threads and by the main thread which detects the problematic groups and splits them into subgroups. So this field changing must be thread safety. Using a critical section here is inconvenient. Therefore we used so-called atomic operations (InterlockedIncrement ()) that allow you to safely increase value of the variable used by multiple threads simultaneously.
The following recursive algorithm is used to determine the group index by the vertex index:
1. The group index of the initial partition is determined by simple dividing the global vertex index on the group vertex number. Then the field subd_state of structure corresponding to the given group is checked. If this value is zero (i.e. the group was not subdivided) then group index is found.
2. If the value is 1 (i.e. the group was split into subgroups) then the next algorithm is used. The group is split on subd subgroups so the subgroup index is determined by the formula:
indx = (iglob - vert_indx) / (vert_number / subd + 1) + first_thread_indx,
where iglob – global vertex index.
3. Now if the subd_state field for the found group is equal to zero then this group was not subdivided and the group index is found. Otherwise the procedure from p. 2. is repeated recursively.

4. Results

The following five scenes were used for algorithm testing:
1. Scene "dispersion"(Fig. 4) consists of 16 objects, 3988110 triangles, 16 parts with different optical attributes, 2002795 vertices and two light sources that are simulated as self-emitting objects. The medium of one objects (prism) has dispersion, its refractive index depends on the wavelength.
2. Scene “room” (Fig. 5) consists of 1 object, 5645608 triangles, 20 parts with different optical attributes, 2925375 vertices and 9 light sources.
3. Scene “vehicle” (Fig. 6) consists of 205 objects, 12533973 triangles, 213 parts with different optical attributes, 6709308 vertices. The scene is illuminated by High Dynamic Range Image and 7 light sources simulated as self-emitting objects.
4. Scene “hall” (Fig. 7) consists of 18207 objects, 387975 triangles, 18207 parts with different optical attributes, 423419 vertices and 361 light sources.
5. Scene “atrium” (Fig. 8) consists of one object, 4543887 triangles, 32 parts with different optical attributes 2380261 vertices and 108 light sources.

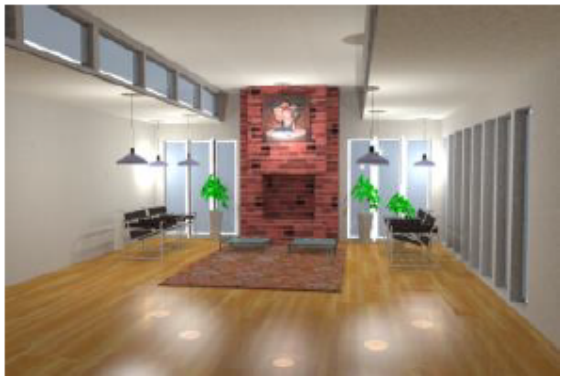


Рис. 5: Test scene room.

Рис. 6: Test scene vehicle.

Рис. 7: Test scene hall.

Рис. 8: Test scene atrium.

The following computers were used during testing:
1. INTEL Xeon E5-2697 v3 – 28 physical cores, with Hyper Threading – 56 threads.
2. INTEL Xeon E7420 – 4 processors, each containing 4 physical cores, Hyper Threading is not supported.
3. INTEL Core (TM) i7-4770 4 physical cores, with Hyper Threading – 8 threads.
Our algorithm efficiency was measured by the speed of ray tracing. The ray speed was measured as the ray-surface intersection per second. The ratios of the speed of our algorithm to the classic one for the scenes and computers described above are shown in the tables 1 (simulation in RGB mode), 2 and 3 (spectral simulation).

Table 1: Speed ratio for simulation in RGB mode.

Scene	1	2	3	4	5
Computer 1	1.92	3.38	1.94	2.05	3.41
Computer 2	0.75	0.79	0.80	0.75	0.75
Computer 3	0.82	0.87	0.88	0.82	0.86

Table 2: Speed ratio for spectral simulation (41 wavelengths).

Scene	1	2	3	4	5
Computer 1	4.43	6.74	2.83	4.45	7.36
Computer 2	2.45	2.69	1.40	2.31	2.61
Computer 3	0.92	1.14	0.92	0.96	1.15

Table 3: Speed ratio for spectral simulation (81 wavelengths).

Scene	1	2	3	4	5
Computer 1	5.49	9.40	3.94	6.35	7.86
Computer 2	2.89	3.41	3.30	3.46	3.52
Computer 3	0.98	1.21	1.05	1.05	1.47

The results show that the proposed parallelization algorithm of forward Monte Carlo ray tracing for Illumination Map calculation on the multi core computers is effective for the modern powerful computers with a large number of physical and virtual cores (e.g. Computer 1 with 28 physical cores). The proposed algorithm provides a significant performance improvement even for relatively simple calculations in RGB mode. In case of spectral calculations it has practically the same efficiency as the classical algorithm even at the relatively slow computers (Computer 3 with four physical and four virtual cores).

5. Conclusion

Suggested algorithm of parallelization of forward Monte Carlo ray tracing provides full loading of physical and virtual cores during Illumination Map calculation on the multi core computers. The solution has significant advantage for spectral and hyper spectral simulation.

Литература

[1] Khodulev A., Kopylov E. Physically accurate lighting simulation in computer graphics software // Proc. 6th International conference on Computer Graphics and Visualization, Russia, 1996, p. 111-119.

[2] Kopylov E., Khodulev A., Volevich V. The Comparison of Illumination Maps Technique in Computer Graphics Software // Proc. 8th International Conference on Computer Graphics and Visualization, Russia, 1998, p. 146-153

[3] Фролов В.А. "Методы решения проблемы глобальной освещенности на графических процессорах"// Диссертация на соискание ученой степени кандидата физико-математических наук по специальности 05.13.11, Москва, ИПМ им. М.В. Келдыша РАН, 2015.

[4] HP company website http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=c04484636

[5] Intel company website https://newsroom.intel.com/wp-content/uploads/sites/11/2016/04/intel-xeon-processor-e5-2600-v4-fact-sheet-x.pdf

[6] R.L.Cook, T.Porter, L.Carpenter. Distributed Ray Tracing. Comp. Graph. (SIGGRAPH'84 Proc.), V.18(3), p.137-145, 1984.