Implementation Details of GPU-based Out-of-Core Many-Lights Rendering

Rui Wang, Yuchi Huo, Yazhen Yuan, Kun Zhou, Wei Hua, Hujun Bao

State Key Lab of CAD&CG, Zhejiang University

In this document, we provide implementation details of the GPU-based out-of-core many-lights rendering method. First, we introduce the organization of out-of-core data and the graph data. Then, we introduce algorithms used in data preparation step. Finally, we give the details of the out-of-core shading step.

1 Out-of-core Data Layout

1.1 Data Layout of Chunks/Blocks

In our algorithm, data are organized in chunks and blocks. Chunks are used in data preparation and blocks are used in shading. Each chunk or block has a 64-bit descriptor, of which 56 bits are used for address, 2 bits indicate the CPU and GPU write locks, 1 bit marks the I/O lock, 1 bit records whether the data is in device or not and 4 bits are reserved for computation in different procedures. For each out-of-core data, we maintain an out-of-core chunk or block array, which stores all source data. In each chunk or block, customized headers are stored for further processing the data. During the computation, when a chunk or block is loaded in the device, the header is first extracted and then the data is processed.

1.2 Data Layout of Hierarchies

For VPL hierarchies, we adapted the storage of the light tree in [Walter et al. 2006] to the GPU. We store energy, position and normal (36 bytes) for each leaf node, and energy, normal direction cone, bounding box, children index and leaf count (64 bytes) for each internal node. Also, an extra 8 bytes per node are used to store the indices of nodes in the hierarchy. These data are organized in a structure-of-arrays (SoA) for better GPU memory access.

For mesh hierarchies, we directly used the data layout in [Stich et al. 2009] and [Pantaleoni and Luebke 2010] to store low-level SBVHs and high-level HLBVHs with triangles.

1.3 Data Layout of the Graph

Fig. 1 illustrates the storage layout of the graph. Graph vertices are stored in two data arrays with two index arrays: a submatrix data array stores graph vertices in the order of submatrix indices and and a mesh data array stores the vertices in the order mesh block indices. Index arrays are used to record the start addresses of each submatrix or mesh block in data arrays. Each graph node is recorded by a 32-bit integer, where 0-12 bits are reserved for mesh blocks and 13-31 bits are for submatrices.

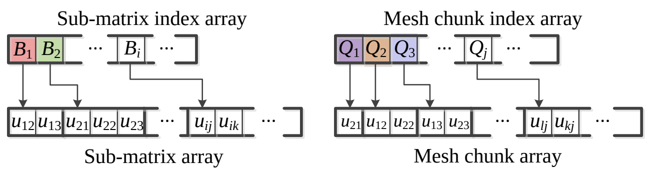
Sub-matrix index array
B_1 B_2 ··· B_i ···
u_{12} u_{13} u_{21} u_{22} u_{23} ··· u_{ij} u_{ik} ···
Sub-matrix array
Mesh chunk index array
Q_1 Q_2 Q_3 ··· Q_j ···
u_{21} u_{12} u_{22} u_{13} u_{23} ··· u_{lj} u_{kj} ···
Mesh chunk array

Figure 1: *An illustration of one submatrix-geometry graph storage.*

Algorithm 1 The out-of-core data preparation algorithm

```
procedure OUTOFCOREDATAPREPARATION( )
    OUTOFCORESORTTRIANGLES()
    while READCHUNK() != EOF
        CONSTRUCTSBVH()
        CREATEANDSTREAMOUTCHUNK()
        //root is the root node of the established SBVH
        $mesh\_representive\_list \leftarrow root$
    end while
    CONSTRUCTHIGHLEVELHLBVH($mesh\_representive\_list$)

    OUTOFCORESORTVPL()
    while READCHUNK() != EOF
        CONSTRUCTKDTREE()
        CREATEANDSTREAMOUTCHUNK()
        //root is the root node of the established KD
        $VPL\_chunk\_list \leftarrow root$
    end while
    CONSTRUCTHIGHLEVELKDTREE($VPL\_chunk\_list$)

    CLUSTERSURFACESSAMPLES()

    PARTITIONMATRIX()

    PACKCHUNKINTOBLOCK()
end procedure
```

2 Out-of-core Data Preparation

Pseudocode is given in Algorithm 1. The construction of these two-level hierarchies takes three steps. First, elements of these two kinds of data, lights or triangles, are sorted according to their spatial Morton code [Pantaleoni and Luebke 2010]. Next, we partition these sorted elements into chunks with similar size in the READCHUNK function. For these chunks, SBVH hierarchies are built for geometry chunks and KD-trees are constructed for lights chunks. Then we pack each chunk with its hierarchy and stream it to device again in the CREATEANDSTREAMOUTCHUNK function. The low-level hierarchies are stored and loaded in-core or out-of-core with their chunk data. Only the bounding boxes of chunks are recorded in a list and used to construct the high-level hierarchy in the CONSTRUCTHIGHLEVELHLBVH and CONSTRUCTHIGHLEVELKDTREE functions. After lights and geometry hierarchies are constructed, we cluster surface samples into a hierarchy by function CLUSTERSURFACESSAMPLES().

Next, the light transport matrix is partitioned into submatrices in the PARTITIONMATRIX function so that each submatrix can be loaded into the device memory all at once. The algorithm is shown in Algorithm 2. The SIMULTANEOUSTRAVERSAL procedure is carried out in a top-down scheme. We start with a lower level of nodes for both hierarchies. Initial pairs of nodes are put into the $active\_list$. In each batch, nodes pairs in $active\_list$ are processed in function SPLITNODES in parallel. To make sure that the active list does not grow too quickly, each time, we only process $n_{\max}$ pairs. In SPLITNODES, given one parent light node, l_j and one parent sample node, s_i, in light and surface sample hierarchies, we split each parent node into two child nodes, p_{i_0}, p_{i_1} and l_{j_0}, l_{j_1} respectively. Then, we compute the bounding shafts between sample children nodes, p_{i_0}, p_{i_1}, and light parent node l_i, and light children nodes, l_{j_0}, l_{j_1}, and sample parent node p_i, respectively. Using these four shafts, s_{i_k} and s_{j_k}, where $k = 0, 1$, we compute all potentially intersected mesh chunks using function COMPUTEPOTENTIALMESHCHUNK. The intersection statuses are stored in bit arrays, b_{i_k} and b_{j_k}, where $k = 0, 1$. If one mesh chunk is potentially intersected, the corresponding bit is set to 1, otherwise 0. We compare the bit difference between one split by a bit operation, exclusive or ($\oplus$), on two bit arrays in function BITCOUNT. The split that producesa larger difference of potentially intersected mesh chunks is chosen as the next traversal direction. New children nodes with the parent node that is not split this time are appended in $active\_list$ for further splits. Once the numbers of potentially intersected mesh chunks of both of the two splits are the same, we stop the split on this pair and output these two nodes to form a submatrix.

After partitioning the matrix, all submatrices are packed into blocks in the PACKCHUNKINTOBLOCK function. Each block is one I/O unit in the shading step and is used to construct the graph. From our simultaneous traversal, lights in one submatrix are from a subtree of one light hierarchy. Thus, we pack each light subtree into one block and further use the light subtree to compute lightcuts in shading. For mesh data, we reuse these bit arrays that indicate the intersection status between submatrices and mesh chunks in the above matrix partition process. For every two mesh chunks, we compare the intersection status with submatrices. If the status of several mesh chunks is exactly the same and the total size of these mesh chunks is able to be loaded into device memory all at once, we combine these mesh chunks into one mesh block. By reorganizing light and mesh data into a more compact data layout, blocks, we are able to control the graph size and reduce data I/O at the shading step.

Algorithm 2 The simultaneous traversal algorithm

```
1:  procedure SPLITNODES($p_i$,$l_j$)
2:      //$p_i$ is a parent node of sample tree
3:      //$l_j$ is a parent node of light tree
4:      $p_{i_0} \leftarrow p_i$.LeftChild; $p_{i_1} \leftarrow p_i$.RightChild;
5:      $l_{j_0} \leftarrow l_i$.LeftChild; $l_{j_1} \leftarrow l_j$.RightChild;
6:      //$s_{i_k}$ and $s_{i_k}$ are bounding shafts
7:      //mesh chunks and bounding shaft
8:      for k=0:1
9:          $s_{i_k} \leftarrow$ COMPUTEBOUNDINGSHAFT($p_{i_k}$, $l_j$)
10:         $s_{j_k} \leftarrow$ COMPUTEBOUNDINGSHAFT($p_i$, $l_{j_k}$)
11:     end for
12:     //$b_k$ are bits indicating intersection status between
13:     //mesh chunks and bounding shaft
14:     for k=0:1
15:         $b_{i_k} \leftarrow$ COMPUTEPOTENTIALMESHCHUNK($s_{i_k}$)
16:         $b_{j_k} \leftarrow$ COMPUTEPOTENTIALMESHCHUNK($s_{j_k}$)
17:     end for
18:     //Compute the difference of bits
19:     $c_0 \leftarrow$ BITCOUNT($b_{i_0} \oplus b_{i_1}$)
20:     $c_1 \leftarrow$ BITCOUNT($b_{j_0} \oplus b_{j_1}$)
21:     //Split the parent node with larger difference
22:     if $c_0 == c_1$ then
23:         OUTPUT($p_i$,$l_i$)
24:     else if $c_0 > c_1$ then
25:         $active\_list$.APPEND($p_{i_0}$,$l_i$)
26:         $active\_list$.APPEND($p_{i_1}$,$l_i$)
27:     else
28:         $active\_list$.APPEND($p_i$, $l_{j_0}$)
29:         $active\_list$.APPEND($p_i$, $l_{j_1}$)
30:     end if
31: end procedure
32:
33: procedure SIMULTANEOUSTRAVERSAL($p_i$,$l_i$)
34:     $active\_list \leftarrow$ INTIALNODES();
35:     while $active\_list$!=NULL
36:         for each ( $p_i$,$l_j$) of $n_{\max}$ pairs in $active\_list$
37:             SPLITNODES($p_i$,$l_j$)
38:         end for
39:     end while
40: end procedure
```

3 Out-of-core Shading

Pseudocode of main out-of-core shading steps is given in Algorithm 3. First, we build up the submatrix-geometry graph in the BUILDGRAPH function, where submatrices are paired with their potentially intersected mesh blocks to composite a vertex. Then, we employ various strategies to find an optimal route, $path\_list$, in SEARCHTRAVELPATH to traverse all vertices. (More details of graph traversal strategies are given in Section 3.1.) Given the optimal path, $path\_list$, the out-of-core shading is iteratively executed to load blocks, compute the lightcuts, test visibilities and cacluate the shading integrations.

In one iteration of the out-of-core shading, a vertex, u_visit, is returned by the NEXTVISIT function from the $path\_list$. If the return is NULL, the whole shading process is terminated. Otherwise, computations of this vertex are performed. We first check if both blocks of u_visit have been loaded in-core by the CHECKBLOCKS function. If not, the computation is blocked and waits. If all data are ready, we compute the lightcuts in SAMPLELIGHTCUTS, test the visibilites in VISIBILITYTEST and take the shading integrations in INTEGRATE. To reduce the latency of waiting for data and overlap some transfers with computations, we use asynchronous data transfer, with a concurrent copy and execute scheme [NVIDIA Corporation 2012]. Specifically, two streams are created and used. All kernels in the SAMPLELIGHTCUTS, VISIBILITYTEST and INTEGRATE functions are launched in one stream, named the computation stream, and data transfer are performed in the other stream, the I/O stream. When the current vertex, u_visit, is being processed, we use the NEXTIO($path\_list$) function to find next vertex, u_load, that requires loading data. If there is enough buffer space, these data of u_load are loaded in the I/O stream by the ASYNCHRONOUSLYLOADBLOCK function. Note that since this I/O runs in a different stream from the one used for computations, the I/O operates asynchronously and does not block the CPU and the GPU. In this way, the I/O cost of loading data for u_load is hidden or partially hidden in computing u_visit. After loading the data in in-core memory, blocks used by u_load are locked in the LOCKBLOCK function in order to prevent them being removed before being used by u_load. If there are no available in-core buffers, the asynchronous I/O data transfers are halted by function HASENOUGHBUFFER until some buffers are released by some computation kernels.

To efficiently utilize the in-core memory buffer, we maintain a Least Recently Used (LRU) list of all in-core blocks. When some new

Algorithm 3 The out-of-core shading algorithm

```
procedure OUTOFCORESHADING
    BUILDGRAPH()
    $path\_list \leftarrow$ SEARCHTRAVELPATH()
    //$u\_visit$ is a vertex being visited.
    //$u\_load$ is a vertex after $u\_visit$ with I/O.
    $u\_visit \leftarrow$ NEXTVISIT($path\_list$)
    SYNCHRONOUSLYLOADBLOCK($u\_visit$)
    LOCKBLOCK($u\_visit$)
    while $u\_visit$ != NULL
        CHECKBLOCK($u\_visit$)
        SAMPLELIGHTCUTS($u\_visit$)
        VISIBILITYTEST($u\_visit$)
        INTEGRATE($u\_visit$)
        $u\_load \leftarrow$ NEXTIO($path\_list$)
        while $u\_load$ != NULL
            if HASENOUGHBUFFER($u\_load$) == FALSE then
                break
            end if
            ASYNCHRONOUSLYLOADBLOCK($u\_load$)
            LOCKBLOCK($u\_load$)
            $u\_load \leftarrow$ NEXTIO($path\_list$)
        end while
        UNLOCKBLOCK($u\_visit$)
        $u\_visit \leftarrow$ NEXTVISIT($path\_list$)
    end while
end procedure
```

submatrix or mesh blocks are scheduled to be loaded into the buffer, we first try to find free spaces in the in-core buffer. If sufficient space is not available, obsolete blocks that are unlocked and least recently used are removed to make the space for new blocks.

3.1 Graph Traversal Strategies

3.1.1 Deterministic Strategies

In Algorithm 4, we give the details of the naive strategy 1, the naive strategy 2 and the MST-based strategy. Because these strategies all assume that the weights of graph edges are deterministic, we name them deterministic strategies. These three static strategies share similar procedures. Initially, we fetch an unvisited vertex u_ij in the graph through the function RANDOM of $unvisited\_list$, and use this vertex as the start of the traversal process. In the traversal step, we append u_ij to $path\_list$, a path list recording the vertices being traversed. Then we remove u_ij from $unvisited\_list$. Next, three approaches are used to search for the next vertex to be visited. In the naive strategy 1, we try to search for the nearest edge among the vertices sharing the same submatrix block B_i. We do this in function SHORTESTEDGE(u_ij, all $\{v\_ix\}$ in $unvisited\_list$). If the return is non-empty, we replace u_ij with the new vertex. If the return is NULL, we turn to searching for the nearest neighbor of u_ij sharing same mesh block Q_j, in function SHORTESTEDGE(u_ij, all $\{v\_xj\}$ in $unvisited\_list$). The searching process continues until all vertices are visited. In the naive strategy 2, the procedure is similar, except that it inverses the searching order such that we first try to find nearest neighbor sharing the same mesh block Q_j, then turn to visit vertices having the same submatrix B_i. In the MST-based strategy, we abandon the constraint on searching order, and attempt to traverse to the nearest neighbor of $\{u\_ij\}$, no matter what submatrix or mesh block it shares.

Algorithm 4 Static strategy

```
//$u$ indicates vertice in the graph
//$u\_ij$ is the one contains submatrix block i and mesh block j
//$path\_list$ is the record of the sequence of visiting vertices

procedure NAIVEONE( )
    $u\_ij \leftarrow unvisited\_list$.RANDOM()
    while $u\_ij$ != NULL
        $path\_list$.APPEND($u\_ij$)
        $unvisited\_list$.REMOVE($u\_ij$)
        while $u\_in \leftarrow$ SHORTESTEDGE($u\_ij$, all $\{u\_ix\}$ in
            $unvisited\_list$) && $u\_in$ != NULL
            $path\_list$.APPEND($u\_in$)
            $unvisited\_list$.REMOVE($u\_in$)
        end while
        $u\_ij \leftarrow$ SHORTESTEDGE($u\_ij$,
        all $\{u\_xj\}$ in $unvisited\_list$)
    end while
    return $path\_list$
end procedure

procedure NAIVETWO( )
    $u\_ij \leftarrow unvisited\_list$.RANDOM()
    while $u\_ij$ != NULL
        $path\_list$.APPEND($u\_ij$)
        $unvisited\_list$.REMOVE($u\_ij$)
        while $u_{nj} \leftarrow$ SHORTESTEDGE($u\_ij$, all $\{u\_xj\}$ in
            $unvisited\_list$) && $u\_nj$ != NULL
            $path\_list$.APPEND($u\_nj$)
            $unvisited\_list$.REMOVE($u\_nj$)
        end while
        $u\_ij \leftarrow$ SHORTESTEDGE($u\_ij$,
        all $\{u\_ix\}$ in $unvisited\_list$)
    end while
    return $path\_list$
end procedure

procedure MST ( )
    $u\_ij \leftarrow unvisited\_list$.RANDOM()
    while $u\_ij$ != NULL
        $path\_list$.APPEND($u\_ij$)
        $unvisited\_list$.REMOVE($u\_ij$)
        $u\_ij \leftarrow$ SHORTESTEDGE($u\_ij$, $unvisited\_list$)
    end while
    return $path\_list$
end procedure
```

3.1.2 Non-deterministic Strategies

The local search strategy and the ant colony strategy update the weights of edges online, thus, we regards them as non-deterministic strategies. The details of local search are given in Algorithm 5. The process of the local search strategy is similar to that of MST-based one. We fetch an unvisited vertex $u\_current$ from the $unvisited\_list$ in RANDOM function and append it to $path\_list$. However, different from the MST-based strategy, we find the next vertex not only with information of $u\_current$, but try to utilize all in-core vertices in the buffer. To be specific, we evaluate a cost for each unvisited vertex in $unvisited\_list$ by summing its distances to all in-core vertices in the DISTANCE function. The vertex with minimum cost for all in-core vertices is selected as the next $u\_current$, and we repeat this process until $unvisited\_list$ is empty. We use a buffer simulator to make use of the dynamic information at run time. The buffer simulator acts similarly as the true buffer except it

Algorithm 5 Local search strategy

```
procedure LocalSearch ( )
    $u\_current \leftarrow unvisited\_list$.Random()
    $path\_list$.Append($u\_current$)
    $unvisited\_list$.Remove($u\_current$)
    while $unvisited\_list$ != EMPTY
        //Swap previous vertices into the buffer simulator.
        $u\_incore\_list \leftarrow$ SwapIn($u\_current$)
        $cost_{min} \leftarrow$ FLT_MAX
        for each $u\_ij$ in $unvisited\_list$
            $cost_{ij} \leftarrow 0$
            for each $u\_nm$ in $u\_incore\_list$
                $cost_{ij}$ += Distance($u\_nm$, $u\_ij$)
            end for
            if $cost_{ij} < cost_{min}$ then
                $u\_current \leftarrow u\_ij$
                $cost_{min} \leftarrow cost_{ij}$
            end if
        end for
        $path\_list$.Append($u\_current$)
        $unvisited\_list$.Remove($u\_current$)
    end while
    return $path\_list$
end procedure
```

does not actually handle memory but emulates a successive virtual space. As showed in the function, we swap the new $u\_current$ into the buffer simulator in function SwapIn, and get back the in-core vertices for the subsequent computation.

Algorithm 6 illustrates our implementation of the ant colony strategy. The core-process of the ant colony strategy involves local search. We use a buffer emulator to imitate dynamic situations, and take the distances to all in-core vertices into account. We utilize the parallelism of the GPU to increase the breadth and depth of our search of solutions in the framework of the ant colony algorithm. Initially, we divide the whole graph into several $subgraph$s to restrict the maximum vertices for each ant, and handle each subgraph in sequence. We launch NUM_ANT threads to search for the optimization solution, and repeat MAX_ITERATIONS times. Each ant-thread initializes its $path\_list$, $unvisited\_list$, traversal cost, and buffer state at the beginning, and then enters the iteration of Append, Remove and finding the next $u\_current$ until the whole $unvisited\_list$ is empty. When we try to find the next $u\_current$, we calculate every unvisited vertex's possibility to be selected according to its distance and phenomenon information. After gathering all p_{ij}, we normalize them by the sum of p_{ij}, and select one based on a random float between 0 to 1 in the RandomSelect function. This selected vertex is the input of the next iteration until the entire subgraph is traversed. Additionally, the cost of each ant is recorded in $path\_list$, which can be fetched in function Cost. The cost is a criteria to update the phenomenon of the graph edge. Ants with a low cost would havea high amount of phenomenon to be distributed in its paths. The phenomenon represents a public information to control the subsequent searching actions, and to drive the ants toward better searching region. Finally, $path\_list$ with lowest cost for each subgraph, $path\_list_{min}$, is appended to $intact\_path\_list$. We reconstruct the buffer states with $intact\_path\_list$ as the beginning of each ant, so we can connect subgraphs as we actually do in the shading process.

3.2 Neighbor Searching in Graph

These graph traversal solutions are designed with different heuristics, but all require an efficient neighbor search. Based on the stor-

Algorithm 6 Ant colony strategy

```
procedure AntColony ( )
    $subgraph\_list \leftarrow$ DivideIntoSubgraph()
    for each $subgraph$ in $subgraph\_list$
        $cost_{min} \leftarrow$ FLT_MAX
        for 1 to MAX_ITERATIONS
            for 1 to NUM_ANT
                //Initialize.
                $unvisited\_list \leftarrow$ all vertices in $sub\_graph$
                $path\_list \leftarrow$ EMPTY
                $cost \leftarrow 0$
                //Continue traversal from previous subgraph.
                $buffer$.Initialize($intact\_path\_list$)

                $u\_current \leftarrow unvisited\_list$.Random()
                $path\_list$.Append
                $unvisited\_list$.Remove($u\_current$)
                while $unvisited\_list$ != EMPTY
                    $u\_incore\_list \leftarrow$ SwapIn($u\_current$)
                    for each $u\_ij$ in $unvisited\_list$
                        $p_{ij} \leftarrow 0$
                        for each $u\_nm$ in $u\_incore\_list$
                            $dist \leftarrow$ Distance($u\_nm$, $u\_ij$)
                            $ph \leftarrow$ Pheromone($u\_nm$, $u\_ij$)
                            //$\alpha$, $\beta$ and $\theta$ are used to control
                            //relative importance of distance and
                            //phenomenon; $\gamma$ is a small number
                            //to prevent dividing by 0.
                            $p_{ij}$ += $(\frac{\theta}{dist+\gamma})^{\alpha} + ph^{\beta}$

                        end for
                    end for
                    //Select a vertice according to possibilities.
                    $u\_current \leftarrow$ RandomSelect($\{p_{ij}\}$)
                    $path\_list$.Append($u\_current$)
                    $unvisited\_list$.Remove($u\_current$)
                end while
                //Add pheromone to edges from active vertices
                //to the one swapped-in at each step.
                UpdatePheromone($path\_list$)
                if Cost($path\_list$) $< cost_{min}$ then
                    $u\_current \leftarrow u\_ij$
                    $cost_{min} \leftarrow$ Cost($path\_list$)
                    $path\_list_{min} \leftarrow path\_list$
                end if
            end for
        end for
        $intact\_path\_list$.Append($path\_list_{min}$)
    end for
    return $intact\_path\_list$
end procedure
```

age formation of our matrix-mesh graph, for a vertex u_{ij}, we first extract the submatrix ID, B_i, and the mesh block ID, G_j, from the node u_{ij}. B_i and G_j are then used to collect all unvisited neighbors, $\{u_{lm}\}$, from the submatrix array and mesh block array, respectively. Based on the storage format of the matrix-mesh graph, these $\{u_{lm}\}$ share one element with u_{ij}, B_i or G_j, thus the I/O costs to swap u_{ij} to them are less than that of any other node. After extractingJ all neighbors of vertices in the active list, we sort these neighbors by the weights and use them in graph traversal.

References

NVIDIA CORPORATION. 2012. *CUDA C Best Practices Guide*, Oct.

PANTALEONI, J., AND LUEBKE, D. 2010. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *High Performance Graphics*, 87–95.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 7–13.

WALTER, B., ARBREE, A., BALA, K., AND GREENBERG, D. P. 2006. Multidimensional lightcuts. *ACM Trans. Graph. 25*, 3, 1081–1088.