In this part, we present how to parallelize quadtree-based recursive tile subdivision for GPUs. Similar to the previous technique, we use AppendStructuredBuffer, CopyStructureCount, and DispatchIndirect for this problem. Additionally, we address the issue of synchronization.
The application of this part is baking light maps. For Agni’s Philosophy, a rasterization-based algorithm was used for offline light map baking. This GPU based technique was inspired by the Parthenon renderer which was developed by Toshiya Hachisuka. This technique generates a set of parallel rays by using GPU rasterization, which is called ray-bundle tracing. To handle many depth fragments for each ray, a per-pixel linked list was used. This was simple and fast for small scenes.
However, for large scenes, this rasterization based ray-bundle tracing had a critical problem. GPU memory is limited for the render target of a ray-bundle. Agni’s Philosophy had scenes that measured a few kilometers. In order to render such vast scenes, the render target must be split into several tiles for each global ray-bundle direction.
Therefore, we introduced an adaptive tiling technique on the GPU for the render target of ray-bundle tracing. This performs quadtree-based recursive tile subdivision according to a lower-resolution importance mipmap. The importance represents required ray density. For our application, it is given by rendering the texel density of light maps. Such tile subdivision was also used for fitted virtual shadow maps, but it was performed on the CPU. In this talk, we present this recursive tile subdivision on the GPU.

In order to build a quadtree implicitly, our importance map resolution is given as a power of two, and the maximum mipmap of the importance map is generated. Once the importance maximum mipmap is built, recursive tile subdivision is started from the top mip level.
This is pseudo code of the recursive tile subdivision.
The input values of this procedure are a mip level and pixel ID of the importance
mipmap.
This function first gets the importance and its threshold value.
The importance represents required buffer resolution per unit area, while the
threshold represents memory capacity.
Therefore, if the importance is smaller than the threshold, the tile information
corresponding to the current mip level and pixel ID is outputted.
Otherwise, the tile is recursively subdivided by descending the mipmap.
This algorithm is optimizable with careful use of synchronizations.
Such quadtree-based recursive algorithms are difficult to implement for GPUs using a single compute pass. However, for each level, processes are independent from each other. Thus, a multi-pass implementation parallelizing for each level is often employed. In such algorithms, a compute pass is launched for each level. Therefore, temporary values, which are required for the next pass, have to be stored in device memory at the end of the pass, and then they have to be restored in the next pass.
This multi-pass algorithm can be implemented using DirectCompute. For each pass, output of the compute kernel is stored in an AppendStructuredBuffer. The next compute pass is dispatched depending on this output. Similar to the point based rendering, it uses CopyStructureCount to obtain the hidden counter value of the AppendStructuredBuffer. Then a single thread compute pass is executed to calculate the number of workgroups for the next level. Finally, the next compute kernel is dispatched using DispatchIndirect which uses input arguments in device memory.
This is the compute shader for the top mip level. For the top mip level, there is only a single node, so only a single thread is dispatched.

If the importance is smaller than the threshold at this level, tile information is appended into an AppendStructuredBuffer. Otherwise, temporary data is outputted using another AppendStructuredBuffer which will be the input for the next level’s compute kernel.
For each level, the upper compute shader is used to calculate the number of workgroups for DispatchIndirect.
And then, the lower shader is dispatched using DispatchIndirect.
This is performed in parallel.
Unlike the top mip level, it first restores the temporary data from the previous level’s kernel.
Although recursive tile subdivision can be performed on the GPU, there are many synchronization points due to the dependency of resources. This synchronization overhead can be more expensive than actual computation, if there are few threads. This is inefficient especially for upper levels. For example, only a single thread is dispatched for the top level, even though a GPU can execute thousands of threads in parallel.
Therefore, we introduce the following optimization technique. In order to reduce the number of synchronization points, several levels of tiles are subdivided in a single compute pass. To perform this, the quadtree is represented using several subtrees. For each subtree, tiles are subdivided in parallel. In this figure, threads are represented with orange arrows.
For this subdivision, a thread is created for each pixel at the finest mip level of the subtree.
Descending the mip level from the top of the subtree, each thread evaluates the subdivision condition.
If the tile is subdivided, the thread descends to the next level.
Otherwise it outputs the final tile information to an AppendStructuredBuffer, and then the algorithm terminates.
Since an identical tile can be evaluated by several threads, the output operation is restricted to a single thread to avoid duplication of tiles.
Although there are many duplicate calculations, especially for the upper nodes of the subtree, this is less expensive than the synchronization overhead if the subtree is not large.
In our experiments, 6 levels were the most efficient size for the subtree.
This is the compute shader for subtrees. As mentioned in the previous slide, a thread is dispatched for each pixel at the finest mip level, and it descends the mip level from the top of the subtree. Descending is done using a simple loop.
This slide shows computation times of the optimized adaptive tile subdivision for different SUBTREE LEVELS.

As previously mentioned, we found setting SUBTREE_LEVELS to 6 to be the fastest, and it is about 6 times faster than the parallelization for each level. For this subtree size, at least 4 K threads run in parallel.

The optimal value for SUBTREE_LEVELS depends on the number of cores in the GPU. If the GPU has a larger number of cores, we can further exploit the parallelism by increasing the subtree size.
To conclude, we have presented a technique for parallelization of recursive tile subdivision utilizing synchronizations. However, too many synchronization points can result in large overhead. To reduce synchronization overhead, subtree-based compute kernels were introduced. Although this subtree-based parallelization performs duplicate calculations, they can be faster than subdivision with many synchronization points.
References