An Implementation of Adaptive Tile Subdivision on the GPU

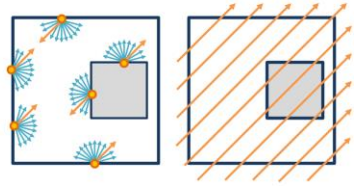
Yusuke Tokuyoshi
Square Enix Co., Ltd.

SA2014.SIGGRAPH.ORG
SPONSORED BY

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.

Application: Baking Light Maps

- Rasterization-based algorithm
  - Inspired by the *Parthenon renderer* [Hachisuka02]
  - Ray-bundle tracing using a per-pixel linked-list [Tokuyoshi11]
  - Used for Agni's Philosophy



Set of parallel rays generated by GPU rasterization

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.

Problem for Vast Scenes

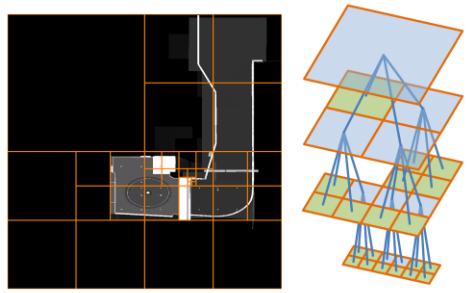
- Limited GPU memory for a render target
- The render target must be split into several tiles

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.

Adaptive Tiling for Ray-Bundles [Tokuyoshi13]

SIGGRAPH ASIA 2014 SHENZHEN

- Quadtree-based **recursive tile subdivision**
  - According to an importance mipmap (light map density)
- Also used for fitted virtual shadow maps [Giegl07]



Subdivided render target of a global ray-bundle

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.

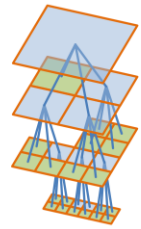
In order to build a quadtree implicitly, our importance map resolution is given as a power of two, and the maximum mipmap of the importance map is generated.
Once the importance maximum mipmap is built, recursive tile subdivision is started from the top mip level.

Recursive Tile Subdivision Algorithm

SIGGRAPH ASIA 2014 SHENZHEN

Algorithm 1 Recursive tile subdivision.

```
procedure RecursiveTileSubdivision( mipLevel, pixelID )
    importance = ImportanceMap[ mipLevel ][ pixelID ]
    threshold = GetThreshold( mipLevel, pixelID )
    if importance < threshold then
        tileData = EncodeTileData( mipLevel, pixelID, importance, threshold )
        Output( tileData )
    else
        RecursiveTileSubdivision( mipLevel - 1, 2 * pixelID )
        RecursiveTileSubdivision( mipLevel - 1, 2 * pixelID + int2( 1, 0 ) )
        RecursiveTileSubdivision( mipLevel - 1, 2 * pixelID + int2( 0, 1 ) )
        RecursiveTileSubdivision( mipLevel - 1, 2 * pixelID + int2( 1, 1 ) )
    end if
```

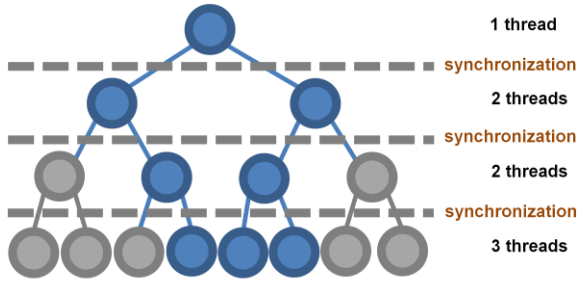


- Optimizable with careful use of synchronizations

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.

Parallelization for Each Level

- Processes at the same level are independent
- Multi-pass implementation
- Often used for tree-based algorithms [Garanzha11, Zhou11, Crassin12]

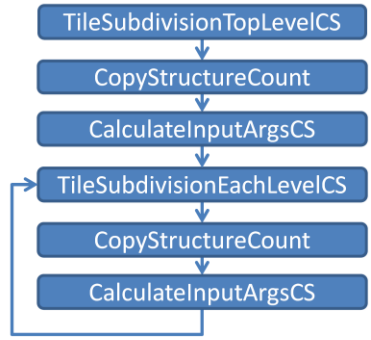


Such quadtree-based recursive algorithms are difficult to implement for GPUs using a single compute pass.
However, for each level, processes are independent from each other.
Thus, a multi-pass implementation parallelizing for each level is often employed.
In such algorithms, a compute pass is launched 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.

Implementation using DirectCompute

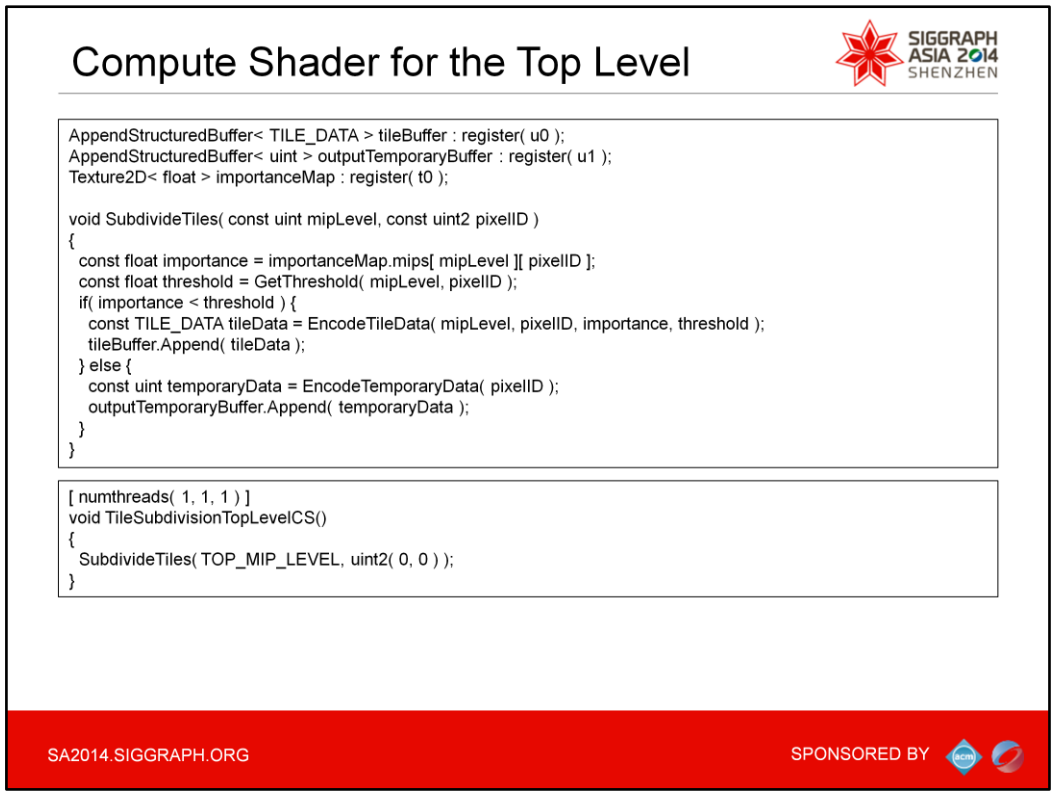
SIGGRAPH ASIA 2014 SHENZHEN

- Output of the compute kernel
  - AppendStructuredBuffer
- Dispatch depending on the previous output
  - CopyStructureCount
  - Calculation of Input Arguments
  - DispatchIndirect



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.

Compute Shader for the Top Level



```
AppendStructuredBuffer< TILE_DATA > tileBuffer : register( u0 );
AppendStructuredBuffer< uint > outputTemporaryBuffer : register( u1 );
Texture2D< float > importanceMap : register( t0 );

void SubdivideTiles( const uint mipLevel, const uint2 pixelID )
{
  const float importance = importanceMap.mips[ mipLevel ][ pixelID ];
  const float threshold = GetThreshold( mipLevel, pixelID );
  if( importance < threshold ) {
    const TILE_DATA tileData = EncodeTileData( mipLevel, pixelID, importance, threshold );
    tileBuffer.Append( tileData );
  } else {
    const uint temporaryData = EncodeTemporaryData( pixelID );
    outputTemporaryBuffer.Append( temporaryData );
  }
}
```

```
[ numthreads( 1, 1, 1 ) ]
void TileSubdivisionTopLevelCS()
{
  SubdivideTiles( TOP_MIP_LEVEL, uint2( 0, 0 ) );
}
```


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 an another AppendStructuredBuffer which will be the input for the next level's compute kernel.

Compute Shaders for Each Level

SIGGRAPH ASIA 2014 SHENZHEN

```
RWBuffer< uint > bufferForArgs : register( u0 );
Buffer< uint > temporaryDataCount : register( t0 );

[ numthreads( 1, 1, 1 ) ]
void CalculateInputArgsCS()
{
  bufferForArgs[ 2 ] = ( temporaryDataCount[ 0 ] + ( WORK_GROUP_SIZE / 4 ) - 1 ) / ( WORK_GROUP_SIZE / 4 );
}
```

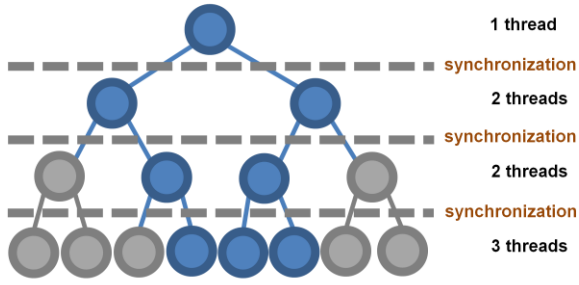
```
StructuredBuffer < uint > inputTemporaryBuffer : register( t1 );
Buffer< uint > temporaryDataCount : register( t2 );

[ numthreads( 2, 2, WORK_GROUP_SIZE / 4 ) ]
void TileSubdivisionEachLevelCS( const uint3 id : SV_DispatchThreadID )
{
  if( id.z >= temporaryDataCount[ 0 ] ) {
    return;
  }
  cosnt uint2 parentPixelID = DecodeTemporaryData( inputTemporaryBuffer[ id.z ] );
  const uint2 pixelID = 2 * parentPixelID + id.xy;
  const uint mipLevel = TOP_MIP_LEVEL - 1 - gPassIndex;
  SubdivideTiles( mipLevel, pixelID );
}
```


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.

Limitations

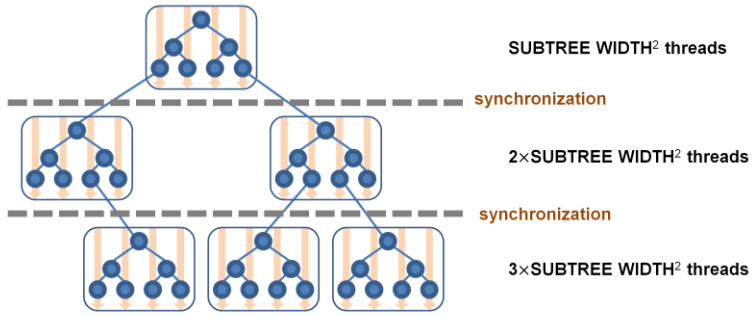
- Many Synchronizations
- Can be a bottleneck especially for upper levels



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.

Reducing Synchronizations

- Subtree-based parallelization
- Several levels of tiles are subdivided in a single pass

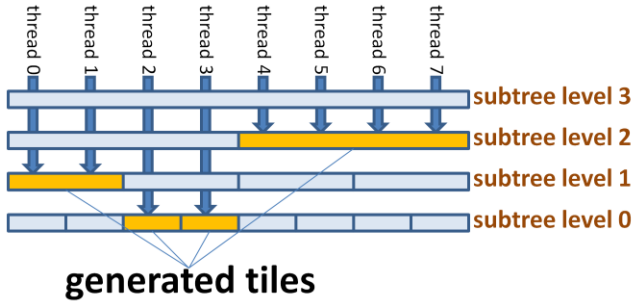


Therefore, we introduce the following optimization technique.
In order to reduce the number of synchronization points, several levels of tiles are subdivided in a single compute pass.
To perform this, the quadtree is represented using several subtrees.
For each subtree, tiles are subdivided in parallel.
In this figure, threads are represented with orange arrows.

Tile Subdivision for the Subtree

SIGGRAPH ASIA 2014 SHENZHEN

- A thread is created for each node at the lowest level of the subtree
- Duplicate calculations for upper nodes
- Efficiency depends on the subtree size
  - 6 levels are the best size in our experiment



generated tiles

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.

Compute Shader for Subtrees

SIGGRAPH ASIA 2014 SHENZHEN

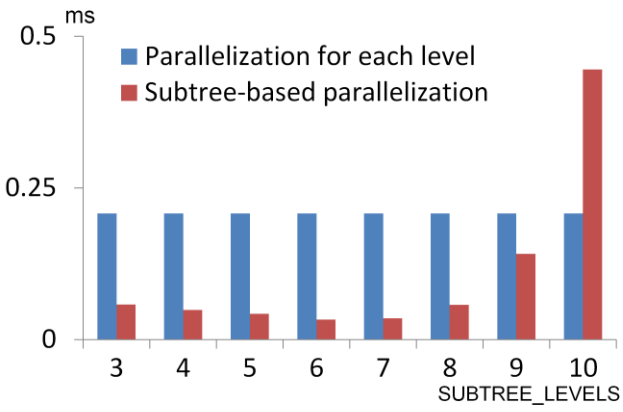
```
AppendStructuredBuffer< TILE_DATA > tileBuffer : register( u0 );
AppendStructuredBuffer< uint > outputTemporaryBuffer : register( u1 );
Texture2D< float > importanceMap : register( t0 );

void SubdivideTilesForSubtrees( const uint upperLevel, const uint lowerLevel, const uint2 id )
{
  for( int i = upperLevel; i > lowerLevel; --i ) {
    const uint width = 1 << ( i - lowerLevel );
    const uint2 pixelID = id / width;
    const float importance = importanceMap.mips[ i ][ pixelID ];
    const float threshold = GetThreshold( i, pixelID );
    if( importance < threshold ) {
      const uint2 outputID = pixelID * width;
      if( id.x == outputID.x && id.y == outputID.y ) {
        const TILE_DATA tileData = EncodeTileData( i, pixelID, importance, threshold );
        tileBuffer.Append( tileData );
      }
      return;
    }
  }
  SubdivideTiles( lowerLevel, id );
}
```

For more details, please refer to the course note

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.

Experimental Results



experimental scene

Averaged computation time for a global ray-bundle direction

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.

Summary

- Parallelization of a recursive tile subdivision utilizing synchronizations
- Many synchronizations can produce a large overhead
- Duplicate calculations can be faster than using many synchronizations

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

- CRASSIN, C., AND GREEN, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. In OpenGL Insights. CRC Press, ch. 22, 303–319.
- GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster hlbvh with work queues. In Proc. HPG '11, 59–64.
- GIEGL, M., AND WIMMER, M. 2007. Fitted virtual shadow maps. In Proc. GI '07, 159–168.
- HACHISUKA T.: Parthenon renderer, 2002. URL: http://www.bee-www.com/parthenon/.
- TOKUYOSHI, Y., SEKINE, T., DA SILVA, T., AND KANAI, T. 2013. Adaptive ray-bundle tracing with memory usage prediction: Efficient global illumination in large scenes. Comput. Graph. Forum 32, 7, 315–324.
- ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2011. Data-parallel octrees for surface reconstruction. IEEE TVCG 17, 5, 669–681.