An Implementation of Adaptive Tile Subdivision on the GPU

Yusuke Tokuyoshi*
Square Enix Co., Ltd.

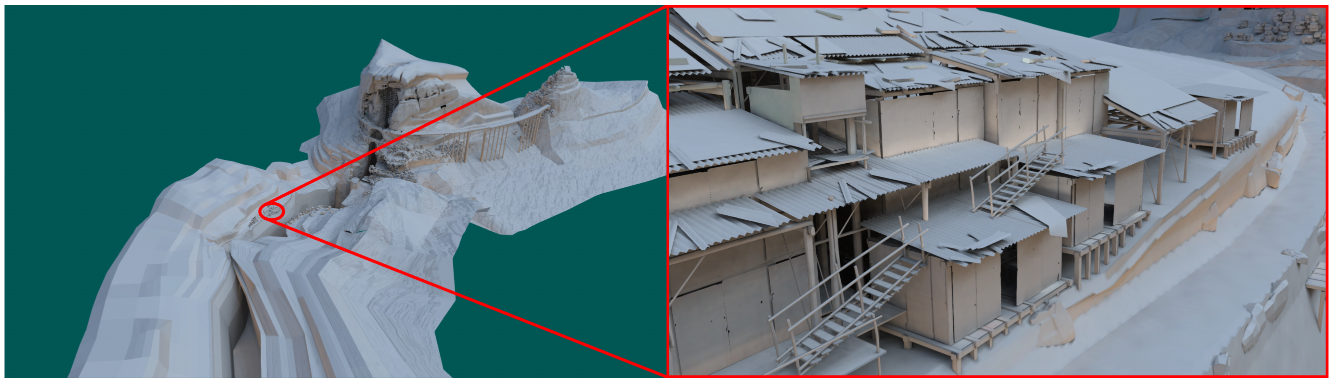

Figure 1: *One of the applications of adaptive tiling is adaptive ray-bundle tracing. This scene has 45.3M texel light maps (4.9 km in diameter, 3.7M triangles). These light maps are precomputed using ray-bundle tracing with adaptive tiling.*

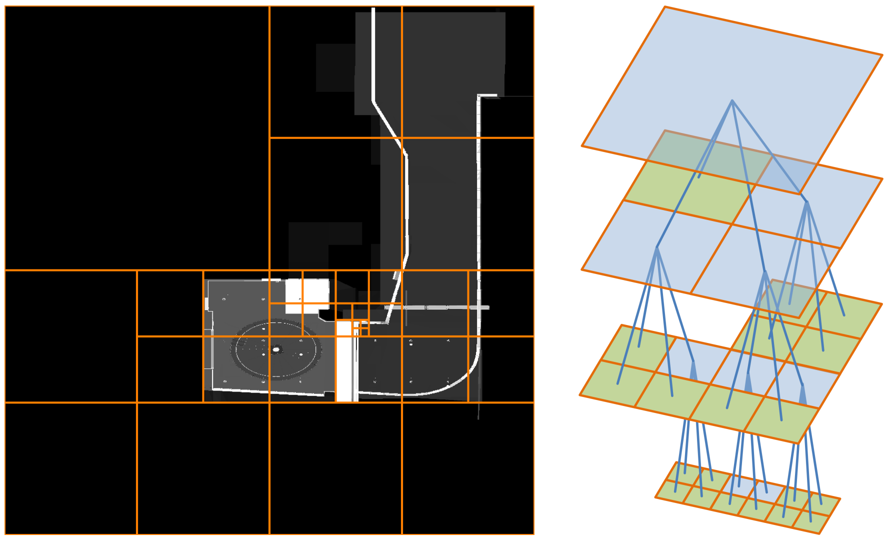

Figure 2: *Adaptive tiling according to importance based on a quadtree (i.e., mipmap of the importance map). Left: importance map viewed from a sample direction. (importance is represented with brightness). For baking light maps using ray-bundle tracing, the importance map is given by rendering light map texel densities. Right: quadtree-based adaptive tile subdivision.*

1 Introduction

In this course, we explain a GPU implementation of adaptive tiling which can be used for several applications such as fitted virtual shadow maps [Giegl and Wimmer 2007] and adaptive ray-bundle tracing [Tokuyoshi et al. 2013] (Fig. 1). For these applications, the render target is split into several tiles adaptively in order to fit an *importance map* which represents required ray densities, as shown in Fig. 2. This adaptive tiling is done by a quadtree-based recursive tile subdivision. For GPUs, it is optimizable with careful use of synchronizations.

In this course note, Sect. 2 first explains a basic algorithm of tile subdivision using a recursion style. Then, Sect. 3 shows a simple parallelization of this tiling using DirectCompute. However, this implementation can have a lot of synchronization overheads. Therefore, we introduce an optimization technique to reduce synchronizations in Sect. 4. Finally, experimental results of this optimization are shown in Sect. 5.

*e-mail:tokuyosh@square-enix.com

2 Algorithm of Adaptive Tile Subdivision

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, the recursive tile subdivision is started from the top mip level. If the importance (e.g., required buffer resolution) for a tile is greater than a given threshold (e.g., memory capacity), the tile is subdivided recursively by descending the mipmaps. Algorithm 1 shows a pseudo code of our tile subdivision. The input parameter *mipLevel* is the current mip level, and the 2D vector value *pixelID* is the pixel position of the importance map at the mip level. If the importance is lower than the threshold, the current tile information (i.e., mip level, pixel position, importance, and threshold) is outputted. Otherwise, this procedure is called recursively for a lower mip level.

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
```

3 Parallelization for Each Level

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 [Garanzha et al. 2011; Zhou et al. 2011; Crassin and Green 2012]. 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.

Algorithm 2 shows a pseudo code of this multi-pass tile subdivision. *TOP_MIP_LEVEL* is the top mip level of the importance mipmap. The procedures *TileSubdivisionTopLevel* and *TileSubdivisionEachLevel* represent compute passes on a GPU which parallely subdivide tiles for each level. Buffers *temporaryBuffer[0]* and *temporaryBuffer[1]* are in device memory, which hold temporary data between consecutive compute passes. They are accessed by ping-pong buffering.

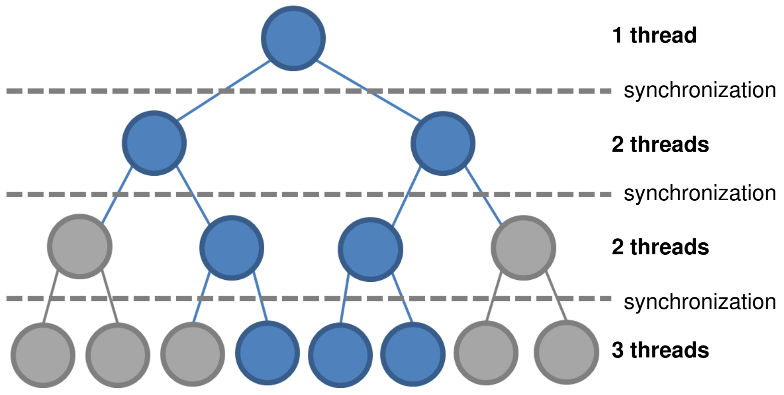

Figure 3: *Parallel tile subdivision for each level.*

Algorithm 2 Multi-pass tile subdivision using parallelization for each level.

```
procedure MultipassTileSubdivision()
  temporaryBuffer[ 0 ] = TileSubdivisionTopLevel()
  for i = 0 to TOP_MIP_LEVEL - 1 do
    temporaryBuffer[ (i + 1) % 2 ] = TileSubdivisionEachLevel( i, temporaryBuffer[ i % 2 ] )
  end for
```

3.1 Implementation Details using DirectCompute

Fig. 4 shows the pipeline using DirectCompute. Each procedure consists of three passes which have a dependency on resources of the previous pass.

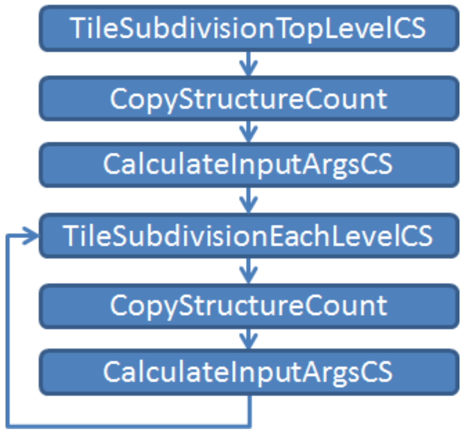

Figure 4: *Pipeline of the parallel tile subdivision using DirectCompute.*

TileSubdivisionTopLevel. Program 1 shows the compute shader of *TileSubdivisionTopLevel* of Algorithm 2. Since the top mip level has only a single pixel, this shader is dispatched with a single thread. If the tile is not subdivided, it is outputted into *tileBuffer* which is an *AppendStructuredBuffer*. Otherwise, the tile information (i.e., *pixelID*) is stored in *outputTemporaryBuffer* which is *temporaryBuffer[0]* of Algorithm 2. Both *temporaryBuffer[0]* and *temporaryBuffer[1]* are also AppendStructuredBuffers. The hidden counter values of *tileBuffer* and *outputTemporaryBuffer* are initialized by $\{0, 0\}$ using *ID3D11DeviceContext::CSSetUnorderedAccessViews*. Once this compute shader is finished, the counter value of *outputTemporaryBuffer* is copied to a buffer *temporaryDataCount* in device memory by using *ID3D11DeviceContext::CopyStructureCount*. Then, a single thread compute pass shown in Program 2 is dispatched for the next procedure. The detail of Program 2 is described in the next paragraph.

Program 1: *Tile subdivision at 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 ) );
}
```

TileSubdivisionEachLevel. Program 3 is the compute shader of *TileSubdivisionEachLevel* of Algorithm 2. This shader performs in parallel. The number of threads in a work group is $(2, 2, \mathit{WORK\_GROUP\_SIZE}/4)$, where *WORK_GROUP_SIZE* is the total size of the work group which is a multiple of the SIMD width (e.g., 32 for NVIDIA® and 64 for AMD). For this case, *id.xy* represents a local index of a child tile, and *id.z* represents an index of *inputTemporaryBuffer* which has parent tile information. Since *id.z* can be larger than the size of *inputTemporaryBuffer* (given by *temporaryDataCount*) to exploit SIMD efficiency, an early return is used. For this compute pass, the number of dispatched work groups must be $(1, 1, \lceil \mathit{temporaryDataCount}/(\mathit{WORK\_GROUP\_SIZE}/4) \rceil)$. This can be done by *ID3D11DeviceContext::DispatchIndirect* without data transfer from GPU memory to CPU memory. In order to calculate the input argument

Program 2: *Calculation of the input argument for compute units.*

```
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 );
}
```

of DispatchIndirect and store it in a buffer, Program 2 is used. This program updates the number of dispatched work groups for the z-axis using *temporaryDataCount*. For the x- and y-axes, since they are always one, they are initialized with one when *bufferForArgs* is created. The constant value *gPassIndex* is given from “i” of Algorithm 2, and used for calculation of the current mip level. For this pass, the hidden counter value of *tileBuffer* is not initialized, but the counter of *outputTemporaryBuffer* have to be initialized with zero. Therefore, $\{-1, 0\}$ are set for initial counter values of CSSetUnorderedAccessViews.

Program 3: *Tile subdivision at each level.*

```
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 );
}
```

3.2 Limitations

As shown in Fig. 4, this parallelization needs three synchronizations for each level due to the dependency of resources. This synchronization overhead can be more expensive than actual computation, if threads are few. This is inefficient especially for upper levels. For example, only a single thread is dispatched for the top mip level, even though a GPU can execute thousands of threads in parallel. The update of the input argument of DispatchIndirect using the single thread compute pass (Program 2) also cannot exploit parallelism. In addition, the hidden atomic counter of *outputTemporaryBuffer* also has a synchronization overhead. In order to reduce these overheads, the next section introduces an optimization.

4 Reducing Synchronizations

In order to reduce synchronizations, tiles are subdivided for several levels in a compute pass. To perform this, the quad-tree is represented using several subtrees as shown in Fig. 5. For each subtree, tiles are subdivided in parallel as shown in Fig. 6. 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 *tileBuffer*, 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 tile duplication. Although there are many duplicate calculations, especially for the upper mip level of the subtree, this is more inexpensive than the synchronization overheads, if the subtree is not large. In our experiments, 6 levels are most efficient for the subtree size.

Algorithm 3 shows the optimized tile subdivision using subtrees. *SUBTREE_LEVELS* is 6 for our case. *TileSubdivisionTopSubtree*, *TileSubdivisionMiddleSubtrees*, and *TileSubdivisionBottomSubtrees* represents compute passes for subtrees. In this algorithm, since the number of iterations is smaller than Algorithm 2, the synchronization overheads are reduced.

4.1 Implementation Details

Fig. 7 shows the optimized pipeline. In addition to reducing the number of iterations, *CalculateInputArgsCS* is omitted unlike Fig 4.

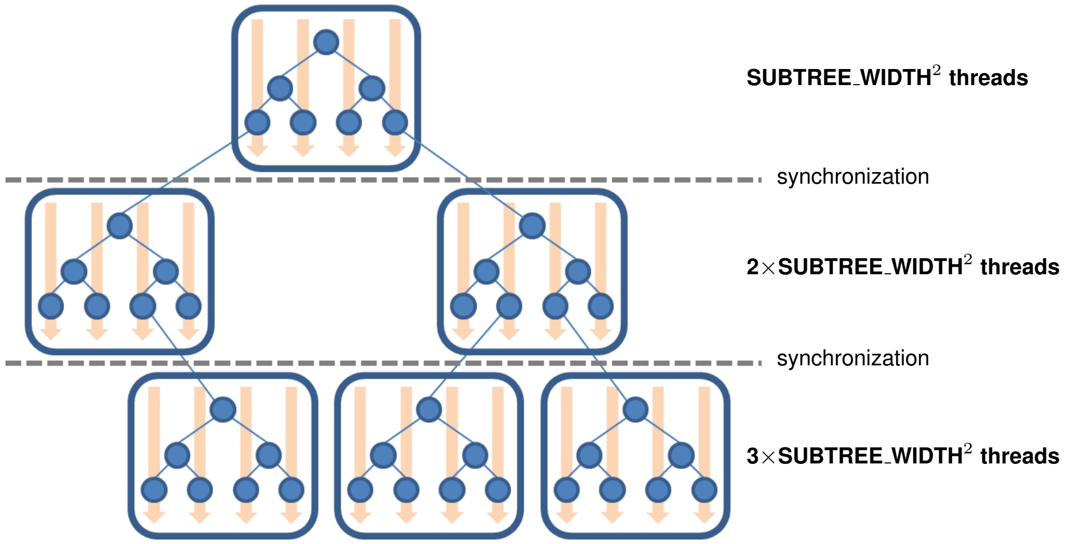

Figure 5: *Subtree-based parallel tile subdivision. SUBTREE_WIDTH^2 threads are dispatched for each subtree (orange arrows).*

Algorithm 3 Optimized tile subdivision using parallelization for each subtree level.

```
procedure OptimizedTileSubdivision()
  temporaryBuffer[ 0 ] = TileSubdivisionTopSubtree()
  N = ⌊TOP_MIP_LEVEL / SUBTREE_LEVELS⌋
  if N > 0 then
    for i = 0 to N - 2 do
      temporaryBuffer[ (i + 1) % 2 ] = TileSubdivisionMiddleSubtrees( i, temporaryBuffer[ i % 2 ] )
    end for
    TileSubdivisionBottomSubtrees( temporaryBuffer[ (N - 1) % 2 ] )
  end if
```

TileSubdivisionTopSubtree. Program 4 is the compute shader of *TileSubdivisionTopSubtree* of Algorithm 3, which performs tile subdivision for the top subtree. The number of dispatched work groups for this shader is $\left(\left\lceil \frac{2^{SUBTREE\_LEVELS}}{WORKGROUP\_WIDTH} \right\rceil, \left\lceil \frac{2^{SUBTREE\_LEVELS}}{WORKGROUP\_WIDTH} \right\rceil, 1\right)$. *WORKGROUP_WIDTH* is given as a power of two, and *WORKGROUP_WIDTH*^{2} should be a multiple of the SIMD width for efficiency. In this course note, $WORKGROUP\_WIDTH = 8$ is used. For this compute pass, the system value *id* represents a pixel position at the finest mip level of the subtree. Unlike the parallelization for each level, *SUBTREE_WIDTH*^{2} (e.g., 64^2) threads are dispatched for the top subtree. As mentioned in the previous paragraph, each thread descends the mip level from the top of the subtree using a simple loop. At the finest mip level of the subtree, this shader computes the *SubdivideTiles* function described in Program 1 which stores temporary data into *outputTemporaryBuffer* for the next procedure.

TileSubdivisionMiddleSubtrees. For subtrees of middle levels, the compute shader shown in Program 5 is used. Similar to Program 3, this shader restores temporary data from *inputTemporaryBuffer* using the system value *id.z*, where *id.z* is the index of the subtree. The pixel index of the top of the subtree is obtained from this temporary data. The system value *id.xy* is a local pixel index at the finest mip level of the subtree. Since the number of threads in a work group for the z-axis is only one, a calculation pass for the input argument of DispatchIndirect is unnecessary unlike Program 2 in Subsect. 3.1. The counter value of the previous *outputTemporaryBuffer* can be copied to *bufferForArgs* directly, using CopyStructureCount with offset 2. Therefore, the overhead caused by Program 2 is now eliminated. In addition, the early termination also can be omitted. The number of work groups for the x- and y-axes are not updated similar to Subsect. 3.1, since they are always $\left\lceil \frac{2^{SUBTREE\_LEVELS}}{WORKGROUP\_WIDTH} \right\rceil$.

TileSubdivisionBottomSubtrees. Program 4 is the compute shader of *TileSubdivisionBottomSubtrees* of Algorithm 3. Since *TOP_MIP_LEVEL* might be undividable by *SUBTREE_LEVELS*, this specialized pass is used for the bottom subtrees.

5 Results

Fig. 8 shows computation times of the optimized adaptive tile subdivision for different *SUBTREE_LEVELS*. For this experiment, tiles are subdivided for adaptive ray-bundle tracing [Tokuyoshi et al. 2013] of the scene of Fig. 2. In our case, $SUBTREE\_LEVELS = 6$ is the fastest, and it is about 6 times faster than the parallelization for each level. For $SUBTREE\_LEVELS = 6$, 4096 threads run in parallel at least. The optimal *SUBTREE_LEVELS* depends on the number of cores in the GPU. If a GPU has a larger number of cores, we can moreover exploit the

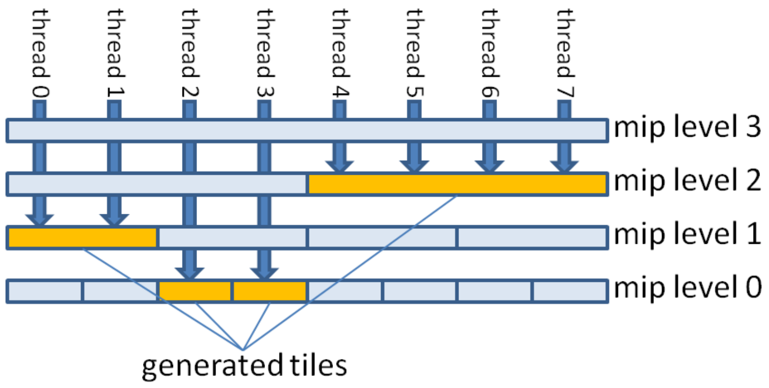

Figure 6: *Synchronization-free tile subdivision for a subtree. A thread is launched for each pixel on the finest mip level of the subtree, and descends the mipmap with duplicate calculations.*

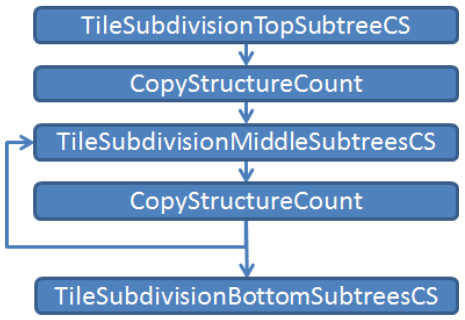
TileSubdivisionTopSubtreeCS
CopyStructureCount
TileSubdivisionMiddleSubtreesCS
CopyStructureCount
TileSubdivisionBottomSubtreesCS

Figure 7: *Pipeline of the optimized parallel tile subdivision.*

parallelism by increasing *SUBTREE_LEVELS*.

6 Conclusion

Dependencies of resources can require a lot of synchronizations. These synchronization overheads are sometimes more computationally expensive than introducing duplicate calculations for GPUs. This course note showed an implementation and optimization of a recursive tile subdivision algorithm. To reduce synchronization overheads, subtree-based compute kernels were introduced. Although these kernels have duplicate calculations, they are amortized by parallelization.

Acknowledgements

The polygon models are courtesy of Square Enix Agni's Philosophy project, Visual Works, and Advanced Technology Division.

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.

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.

Program 4: *Tile subdivision for the top subtree.*

```
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( uint 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 );
}

static const uint LOWER_LEVEL = max( TOP_MIP_LEVELS - SUBTREE_LEVELS, 0 );
static const uint SUBTREE_WIDTH = 1 << min( SUBTREE_LEVELS, TOP_MIP_LEVELS );
static const uint CLAMPED_WORKGROUP_WIDTH = min( WORKGROUP_WIDTH, SUBTREE_WIDTH );

[ numthreads( CLAMPED_WORKGROUP_WIDTH, CLAMPED_WORKGROUP_WIDTH, 1 ) ]
void TileSubdivisionTopSubtreeCS( const uint2 id : SV_DispatchThreadID )
{
  SubdivideTilesForSubtrees( TOP_MIP_LEVEL, LOWER_LEVEL, id );
}
```

Program 5: *Tile subdivision for middle level subtrees.*

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

static const uint SUBTREE_WIDTH = 1 << SUBTREE_LEVELS;

[ numthreads( WORKGROUP_WIDTH, WORKGROUP_WIDTH, 1 ) ]
void TileSubdivisionMiddleSubtreesCS( const uint3 id : SV_DispatchThreadID )
{
  const uint upperLevel = TOP_MIP_LEVEL - SUBTREE_LEVELS - gPassIndex * SUBTREE_LEVELS;
  const uint lowerLevel = upperLevel - SUBTREE_LEVELS;
  const uint2 parentPixelID = DecodeTemporaryData( inputTemporaryBuffer[ id.z ] );
  const uint2 pixelID = SUBTREE_WIDTH * parentPixelID + id.xy;
  SubdivideTilesForSubtrees( upperLevel, lowerLevel, pixelID );
}
```

Program 6: *Tile subdivision for bottom subtrees.*

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

static const uint UPPER_LEVEL = TOP_MIP_LEVEL - ( TOP_MIP_LEVEL / SUBTREE_LEVELS ) * SUBTREE_LEVELS;
static const uint SUBTREE_WIDTH = 1 << UPPER_LEVEL;
static const uint CLAMPED_WORKGROUP_WIDTH = min( WORKGROUP_WIDTH, SUBTREE_WIDTH );

[ numthreads( CLAMPED_WORKGROUP_WIDTH, CLAMPED_WORKGROUP_WIDTH, 1 ) ]
void TileSubdivisionBottomSubtreesCS( const uint3 id : SV_DispatchThreadID )
{
  const uint2 parentPixelID = DecodeTemporaryData( inputTemporaryBuffer[ id.z ] );
  const uint2 pixelID = SUBTREE_WIDTH * parentPixelID + id.xy;
  SubdivideTilesForSubtrees( UPPER_LEVEL, 0, pixelID );
}
```

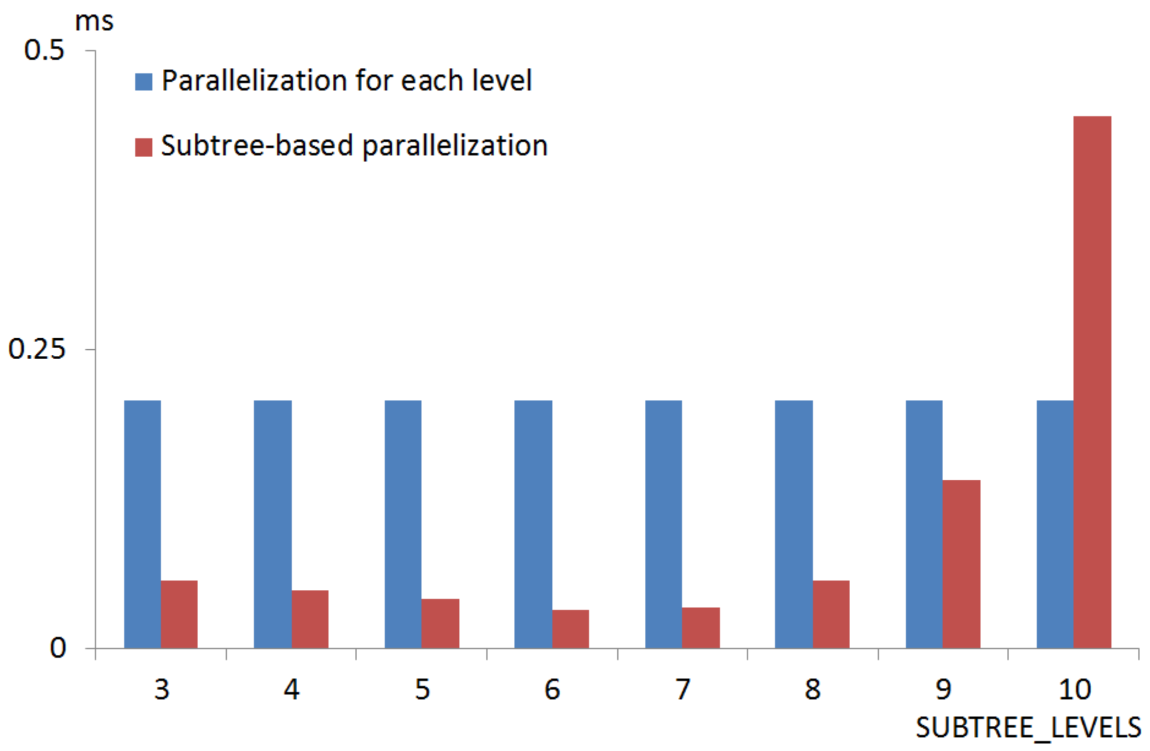

Figure 8: *Average computation times of adaptive tile subdivision for ray-bundle tracing of the experimental scene (Fig .2, importance map resolution: 1024^2, GPU: AMD Radeon^{TM} HD 6990). Red bars are the parallelization for each level described in Sect. 3. Blue bars are the subtree based parallelization described in Sect. 4.*