ForwardPlus11

December 2012, Jason Stewart

AMD
The future is fusion

This is a brief overview of Forward+, which extends classic forward rendering to allow a high number of dynamic lights to be easily supported while retaining the advantages of forward rendering such as “automatic” MSAA support, high performance, and ease of implementation.

Agenda

- Motivation
- Forward+ Algorithm
- Performance

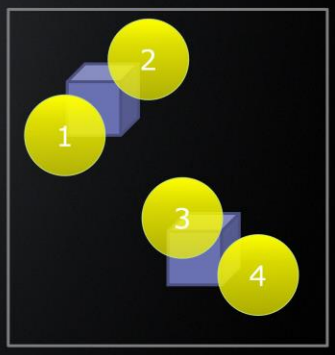
AMD
The future is fusion

Motivation


This example shows 2000 animated lights.
RADEON
GRAPHICS
AMD

Want lots of dynamic lights

- Forward rendering doesn't scale well with high numbers of dynamic lights
- Need CPU-side light management code to determine which lights hit each object
- Each unique set of lights requires changing shader constants
  - And thus requires a new draw call
  - This increases the number of draw calls
  - And it reduces opportunities for instancing



AMD The future is fusion

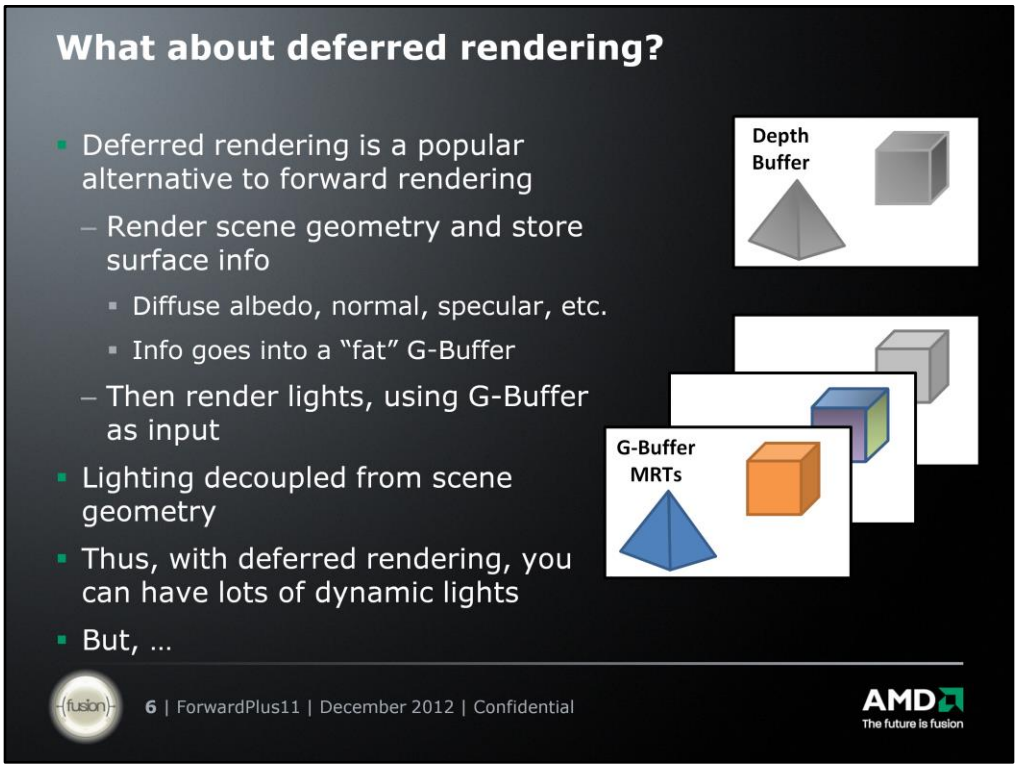
What problem are we trying to solve? We want to support a lot of lights, including dynamic lights. Traditional forward rendering runs into trouble here.

Want lots of dynamic lights

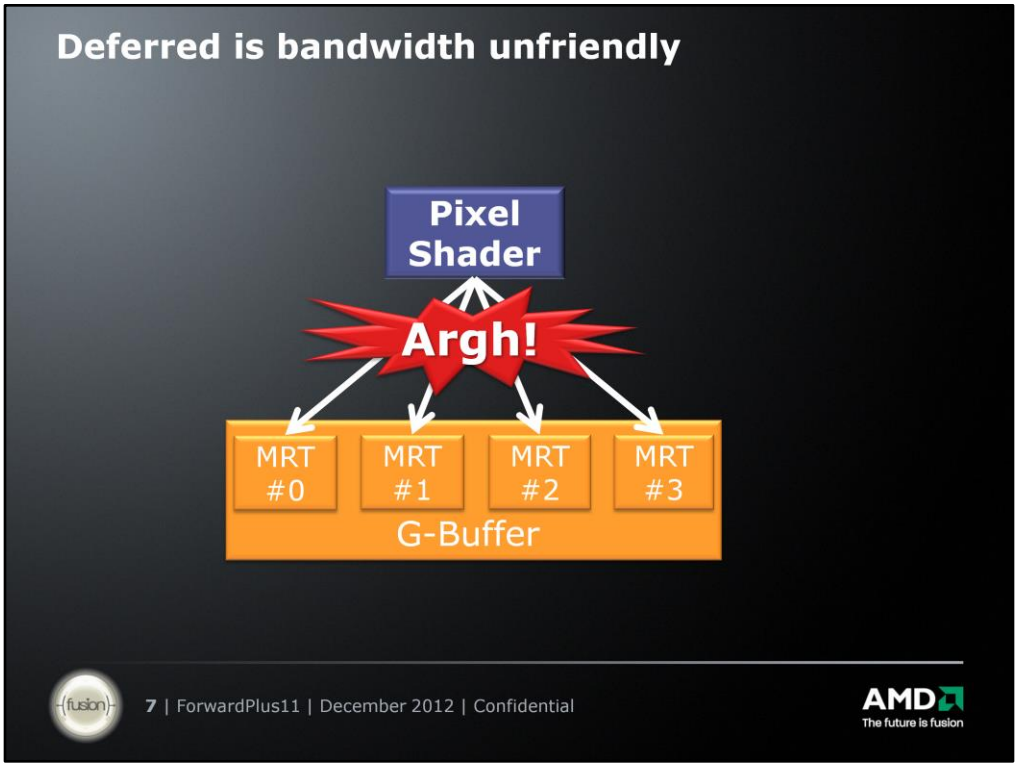
- Culling lights per object is suboptimal
  - Poor granularity
  - Some objects are big
    - e.g. terrain
  - Can bust up large objects into smaller ones
    - But this increases the number of draw calls

AMD
The future is fusion

e.g. for terrain, a particular light may only affect a small part of the object, but still have to process that light at every pixel visible for that object. You can bust up larger objects into smaller ones as part of your art pipeline, but this increases the draw call count. And even relatively small objects are still large from the perspective of light culling efficiency.



Deferred rendering allows a high number of lights. That is, it solves the problem we are trying to solve, and it has gained popularity for that reason. But while it solves our problem, it creates others.



GPUs can be bottlenecked by “export” cost. Export cost is the cost of writing PS outputs into the render targets.

Common scenario as PS is typically short for this pass.

If MSAA is enabled, then each of these RTs is at MSAA resolution, further increasing export cost.

One thing to note here is that ALU power is increasing faster than bandwidth. That is, the ALU/Mem ratio has increased a lot and will continue to increase. So, trading a reduction in ALU operations for an increase in memory bandwidth consumption is not a good trade on modern GPUs.

Deferred is complicated

- MSAA takes a lot of care and feeding
- Still need forward rendering for transparency
- Lots of optimizations needed
  - G-Buffer attribute packing to reduce bandwidth
  - Early depth/stencil culling optimizations during light volume rendering
  - MSAA edge detection
  - There's an entire 40+ page presentation just on deferred rendering optimizations

AMD
The future is fusion

What's good about forward rendering?

- Material variety
- MSAA is "automatic"
- Bandwidth friendly
- Supports transparency
- Straightforward implementation
- Can we solve the "lots of lights" problem?

AMD
The future is fusion

Forward+ Algorithm

AMD
The future is fusion

Classic forward rendering

- Depth pre-pass
  - Fills z buffer
    - Prevents overdraw when shading

- Forward shading
  - Pixel Shader
    - Iterates through light list **set for each object**
    - Evaluates material
      - Diffuse texture, spec mask, bump map, etc.

AMD
The future is fusion

Forward+ rendering

- Depth pre-pass
  - Fills z buffer
    - Prevents overdraw when shading
    - Provides min and max depth per tile during light culling
- **Tiled light culling**
  - **Compute Shader**
  - **Generates per-tile light list**
- Forward shading
  - Pixel Shader
    - Iterates through light list **calculated by tiled light culling**
    - Evaluates material
      - Diffuse texture, spec mask, bump map, etc.

Depth pre-pass

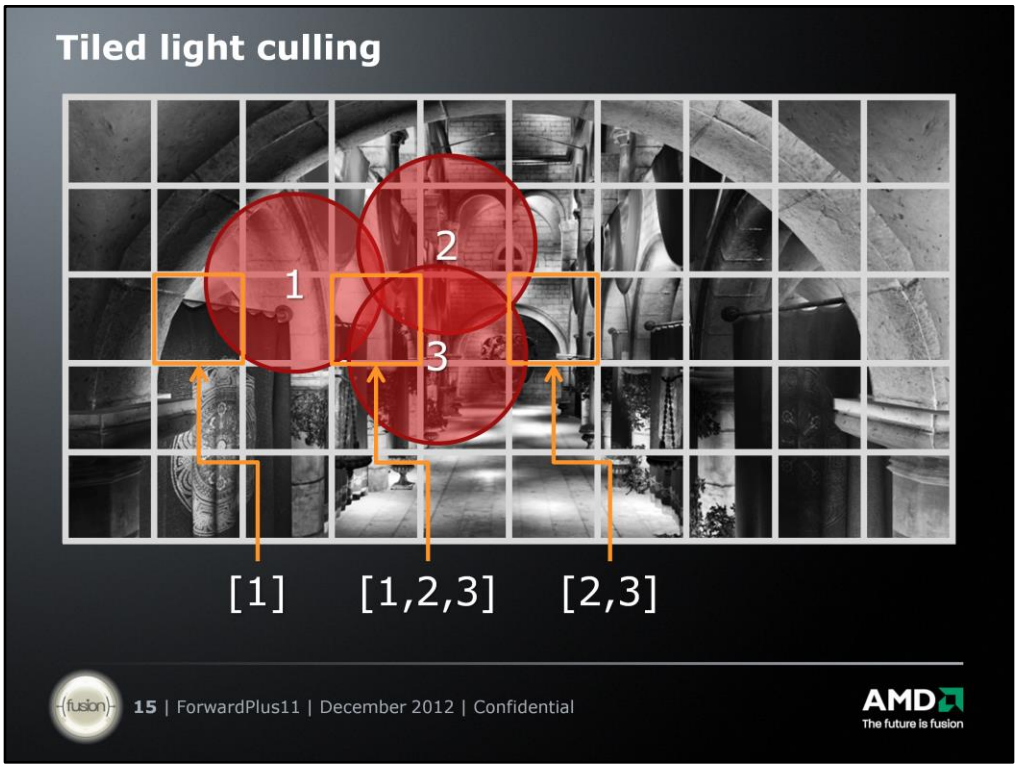
- Prevents overdraw when shading
  - Don't execute the pixel shader for a pixel that gets overwritten later
- Provides min and max depth per tile during light culling
  - Front and back of per-tile frustum for culling
  - Could just use view frustum near and far planes
    - But this results in many false-positive intersections
    - Bad for performance

AMD
The future is fusion

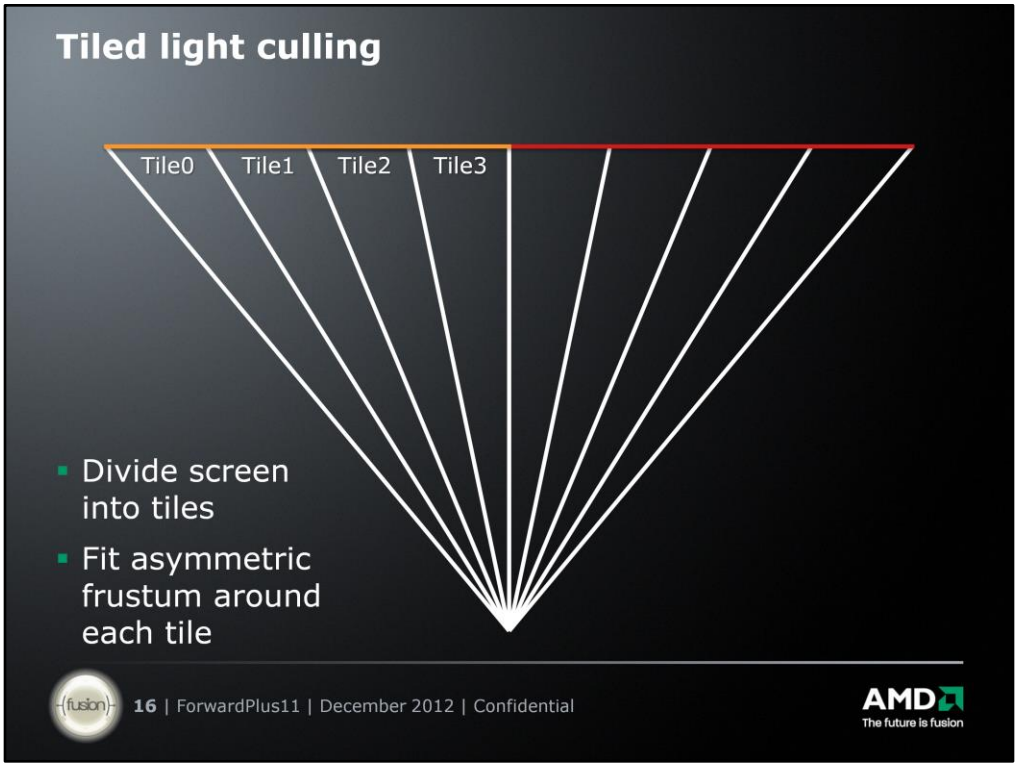
Depth pre-pass

- The usual advice applies
  - Turn off color writes
    - Depth-only rendering is much faster
  - Remember to optimize your geometry
    - Vertex throughput more likely to be a bottleneck in depth-only rendering
    - e.g. D3DXOptimize[Faces][Vertices]
  - Use position-only streams
    - Or position and tex coord for alpha test
    - Improves reuse in pre-VS cache
  - Render front to back

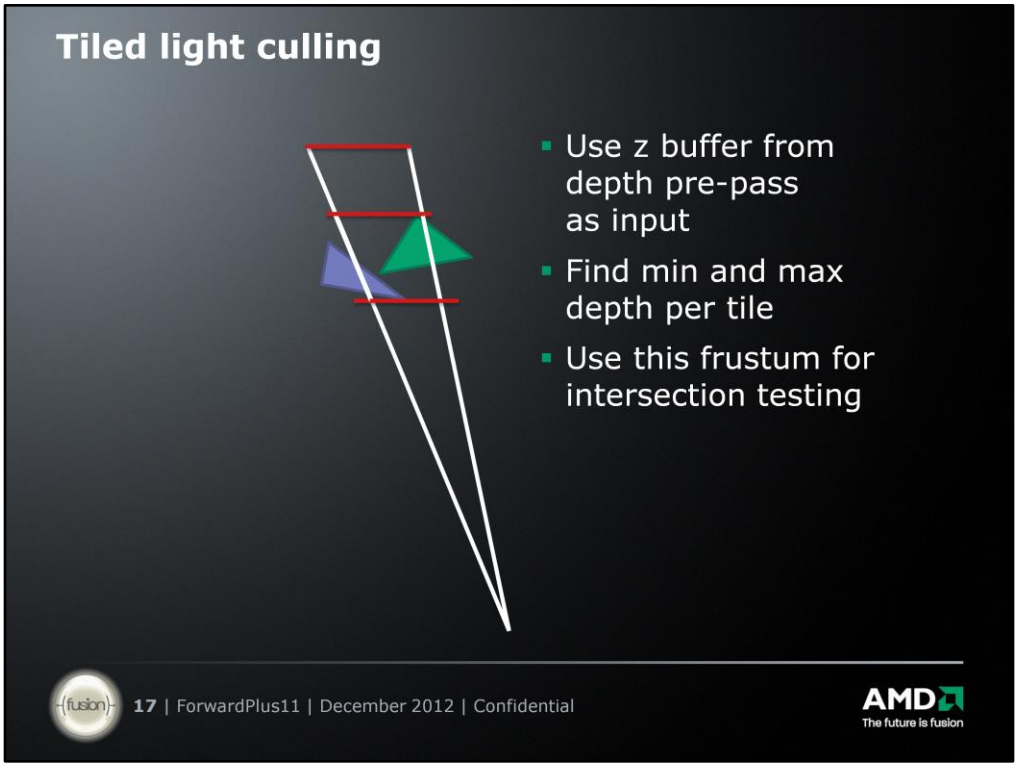
AMD
The future is fusion



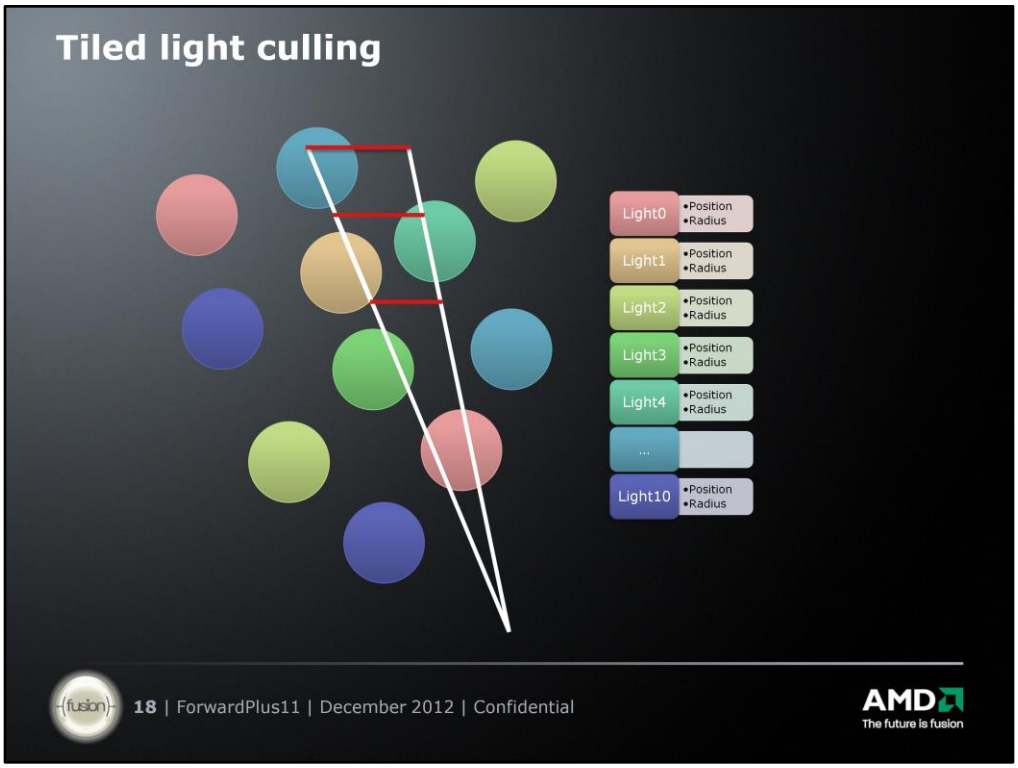
Tiled light culling is the core of the Forward+ algorithm. Organize the screen into a grid of fixed-size tiles. Use a Compute Shader to create a per-tile list of lights.



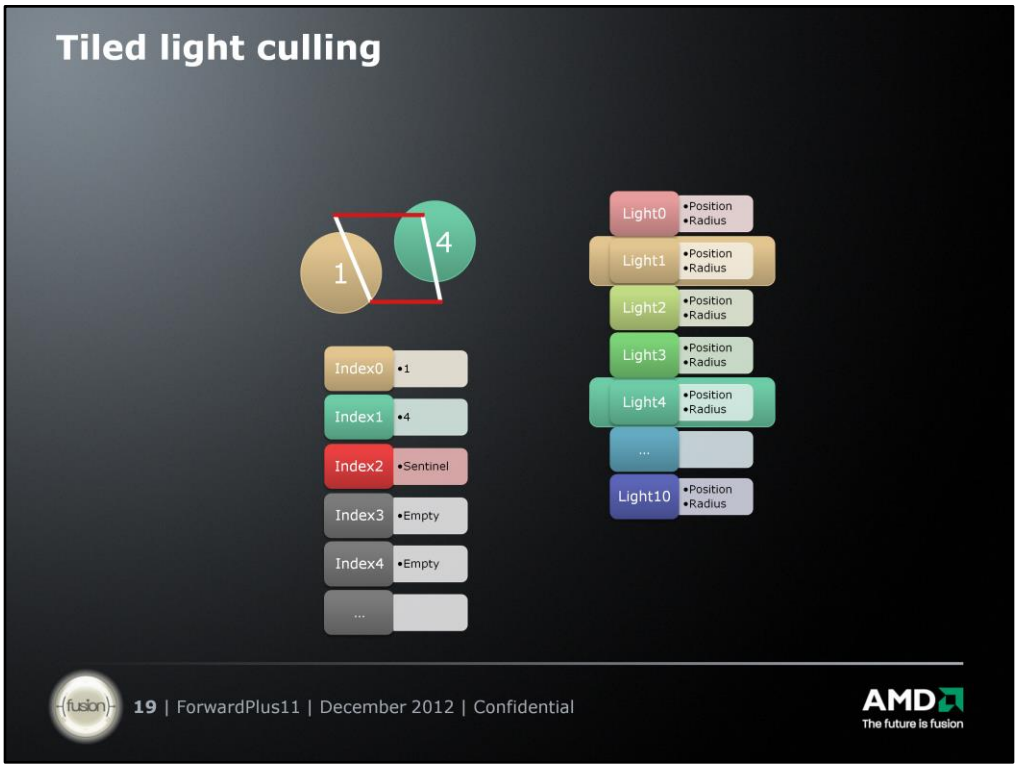
This is a top-down view, in 2D. The screen is represented by the line at the top. We fit a frustum around each tile in view space.



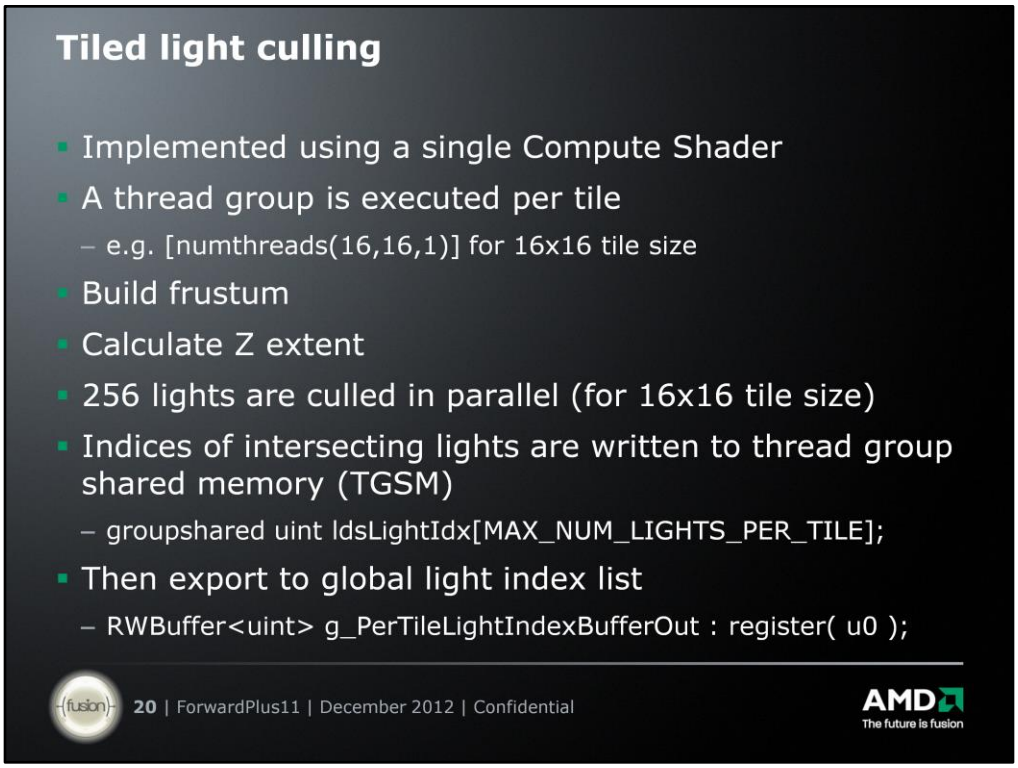
And then, to make the frustum fit more tightly around our scene geometry for the current frame, we use the min and max depth in that tile (from the depth pre-pass) to form the front and back of the frustum.



Then, test each light against the frustum (bounding volume vs. frustum culling).



For each light that intersects the frustum, write the index of that light in the per-tile list. Mark the end of the list with a sentinel.



Thread Group Shared Memory (TGSM) is Direct Compute lingo.

You may also see this type of memory referred to as Local Data Store (LDS) or Thread Local Storage (TLS).

Forward shading

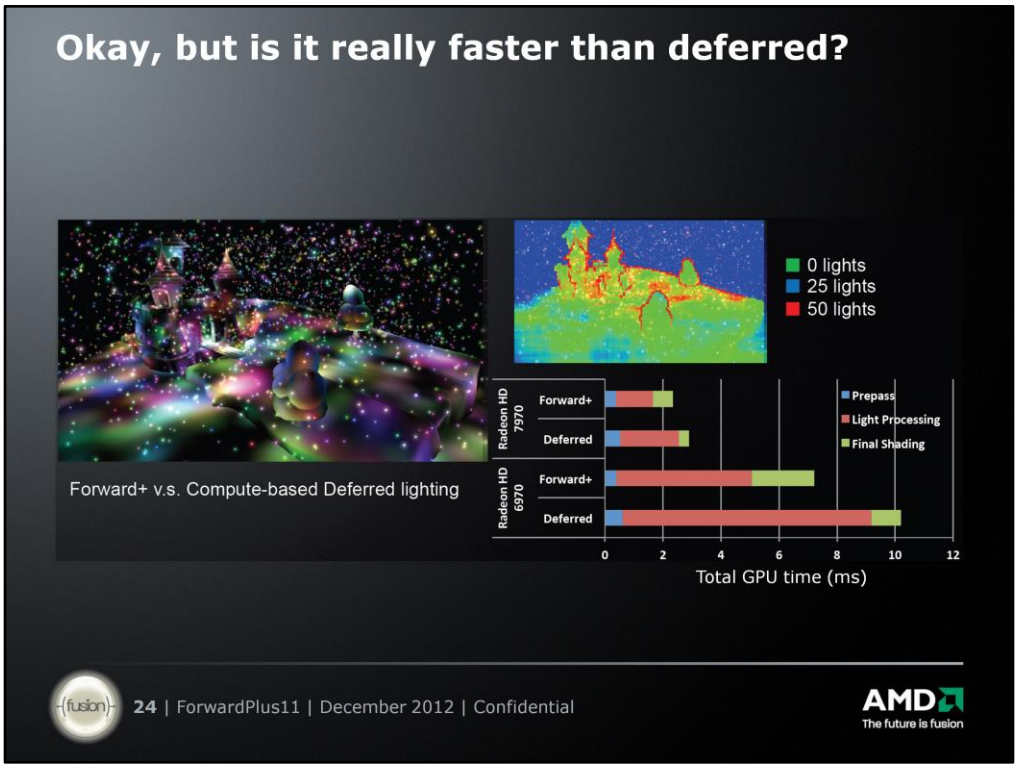
- Forward shading for Forward+ is the same as classic forward rendering
- Except that the light list is now per tile from the Compute Shader
  - Instead of per object in shader constants

AMD
The future is fusion

Performance

But is it fast?

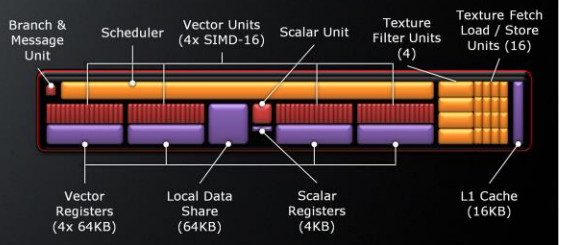
- 1080p, Radeon HD 7970 GHz Edition
- MSAA disabled
  - Culling 1024 lights takes 0.46 ms
  - Culling 3072 lights takes 0.74 ms
- 4x MSAA
  - Culling 1024 lights takes 0.64 ms
  - Culling 3072 lights takes 0.93 ms
- Total frame time will depend on the expense of your light loop during forward rendering
  - For the ForwardPlus11 sample, 2.33 ms total frame time for 1024 lights, 4x MSAA



Benchmark using 3072 dynamic lights. This is using the Leo demo framework, not our DXSDK-style sample. For a direct comparison of Forward+ vs. deferred in our sample framework, see the TiledLighting11 sample.

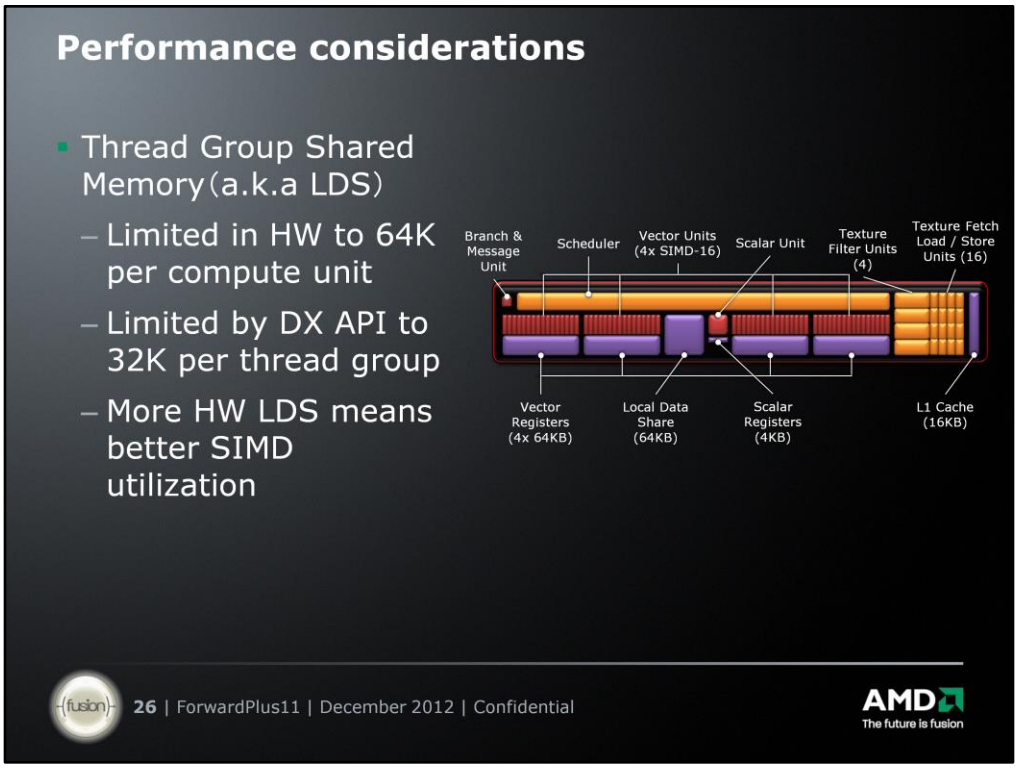
Performance considerations

- Thread groups
  - Threads are always executed in wavefronts on each SIMD
  - Thread group size should be a multiple of the wavefront size (64)
    - Otherwise, [(Thread Group Size) mod 64] threads go unused



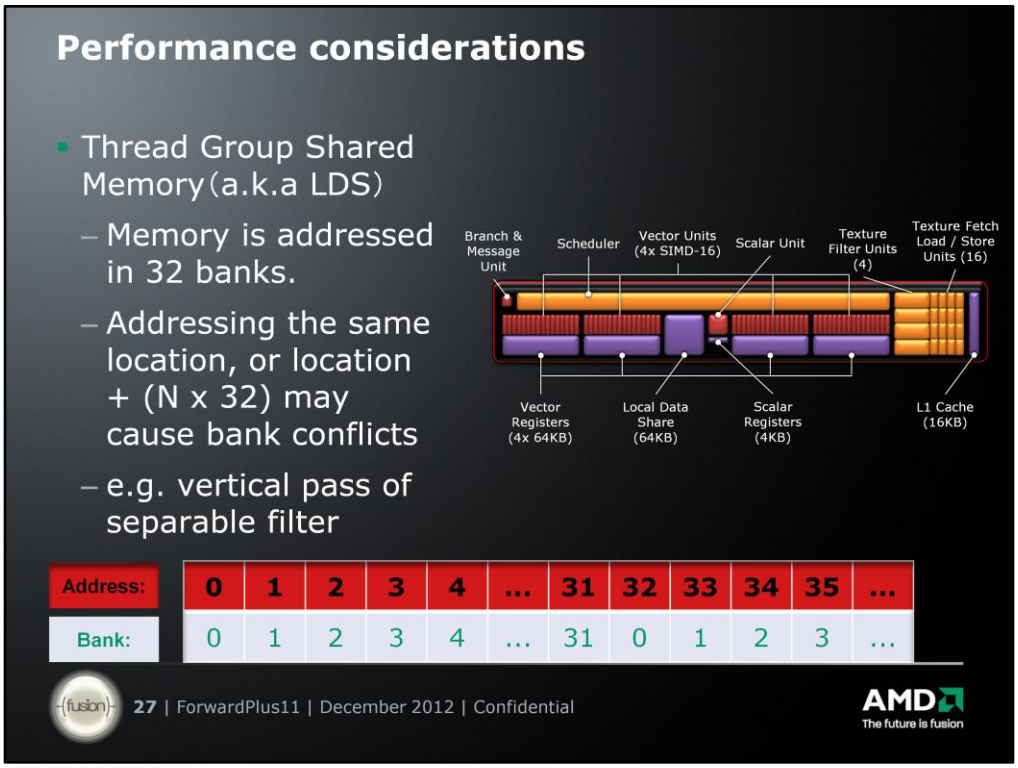
AMD The future is fusion

The diagram is of a Graphics Core Next (GCN) Compute Unit (CU). Each SIMD in the CU is 16 wide. Over 4 cycles, a SIMD processes a “wavefront”. 16 threads per cycle times 4 cycles equals 64 threads in a wavefront. This is where the “multiple of 64” rule comes from for numthreads declarations in a Compute Shader.



Remember, thread group shared memory (TGSM) is DirectX terminology. Local Data Share (LDS) is GCN architecture terminology. When talking about Compute Shaders, LDS is used to implement TGSM.

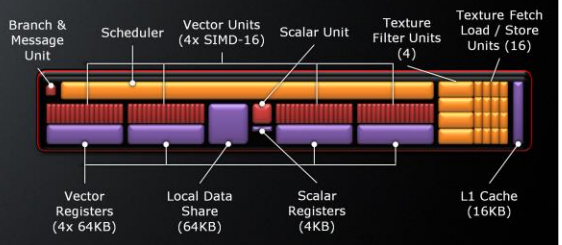
If DirectX limits us to 32K, then why have 64K? Well, that 64K is shared amongst the four SIMDs. If your DirectX Compute Shader uses 16K or less of TGSM, then each of the four SIMDs can be running different thread groups from your Compute Shader dispatch simultaneously, fitting the TGSM of the four different thread groups into LDS.

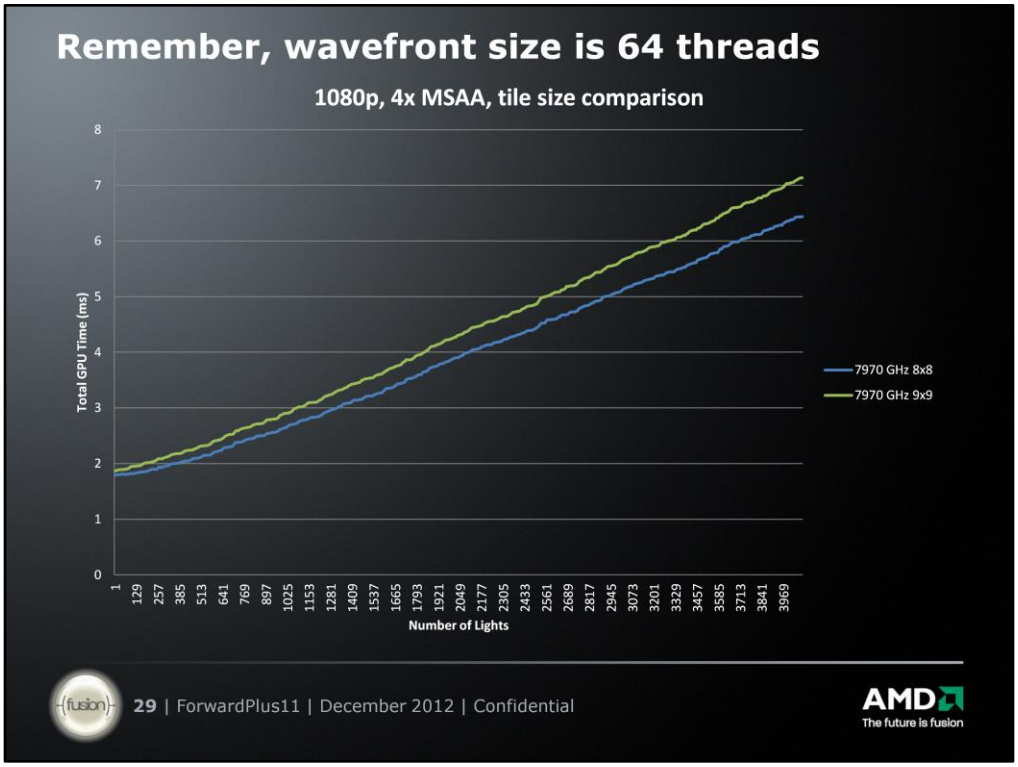


Watch out for bank conflicts. There is a counter that you can view in GPU PerfStudio 2.

Performance considerations

- Ideal thread group configuration? It depends
  - Depends on LDS memory required by your algorithm
  - Should always be a multiple of 64, though
  - Often, more is better, to a point
  - 256 is often a good answer
  - Experiment



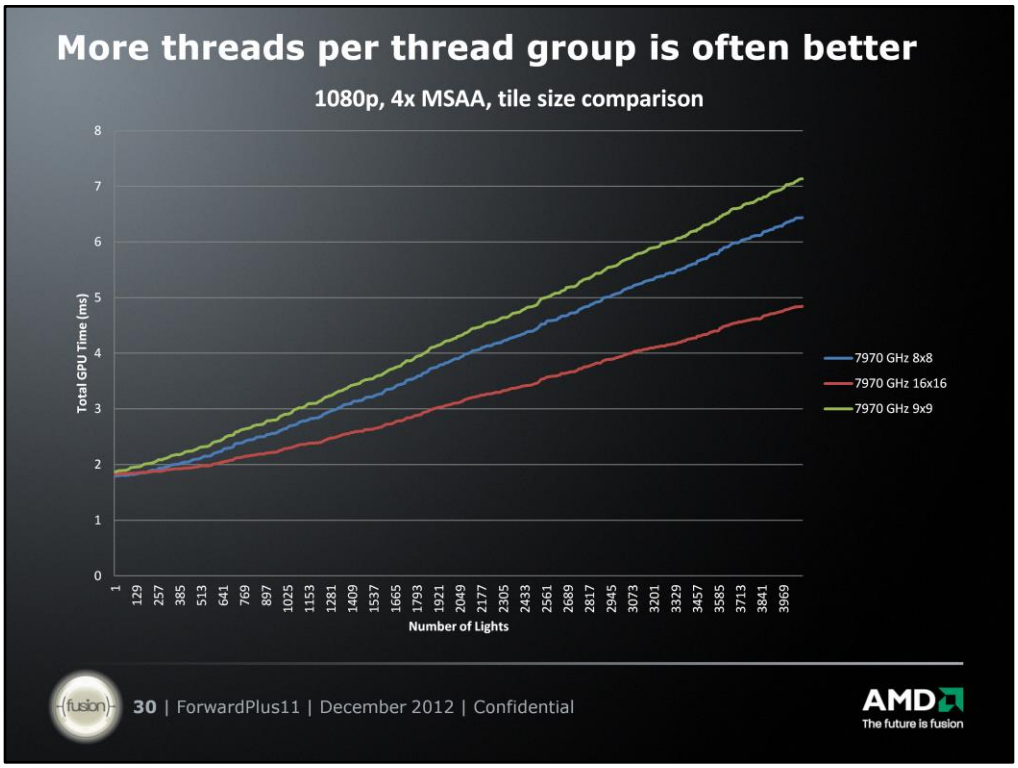


We will now show some real-world performance results related to the performance considerations of the previous slides.

Thread group size should be a multiple of the wavefront size (64). Otherwise, [(Thread Group Size) mod 64] threads go unused.

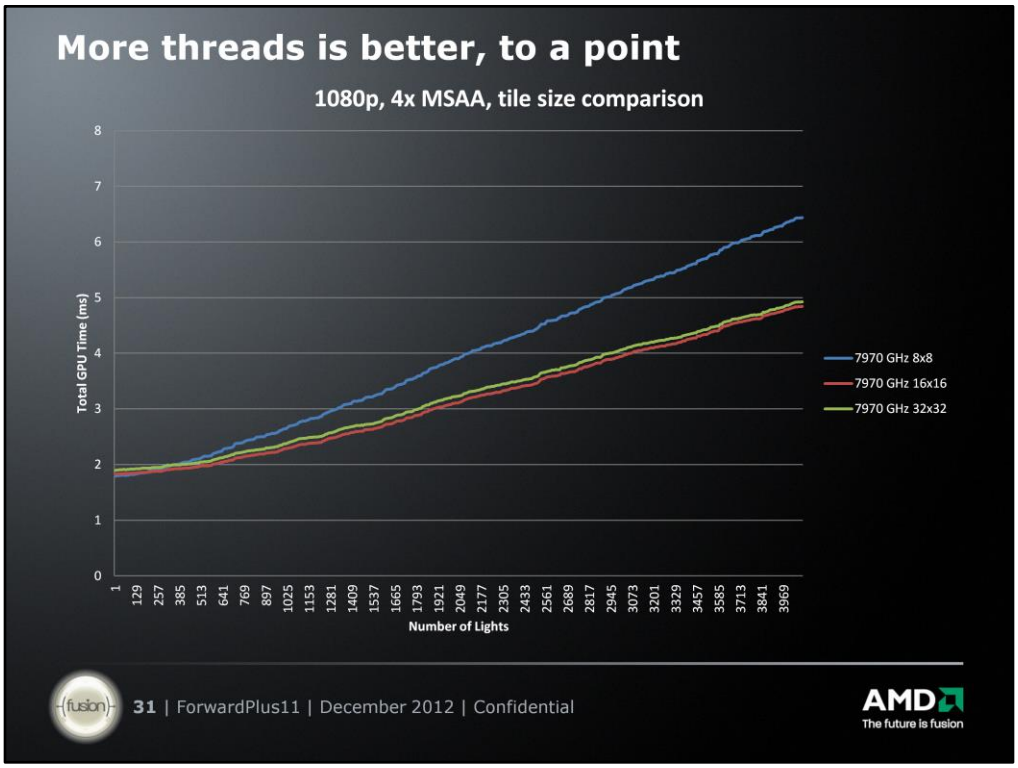
8x8=64 (good)

9x9=81 (bad)

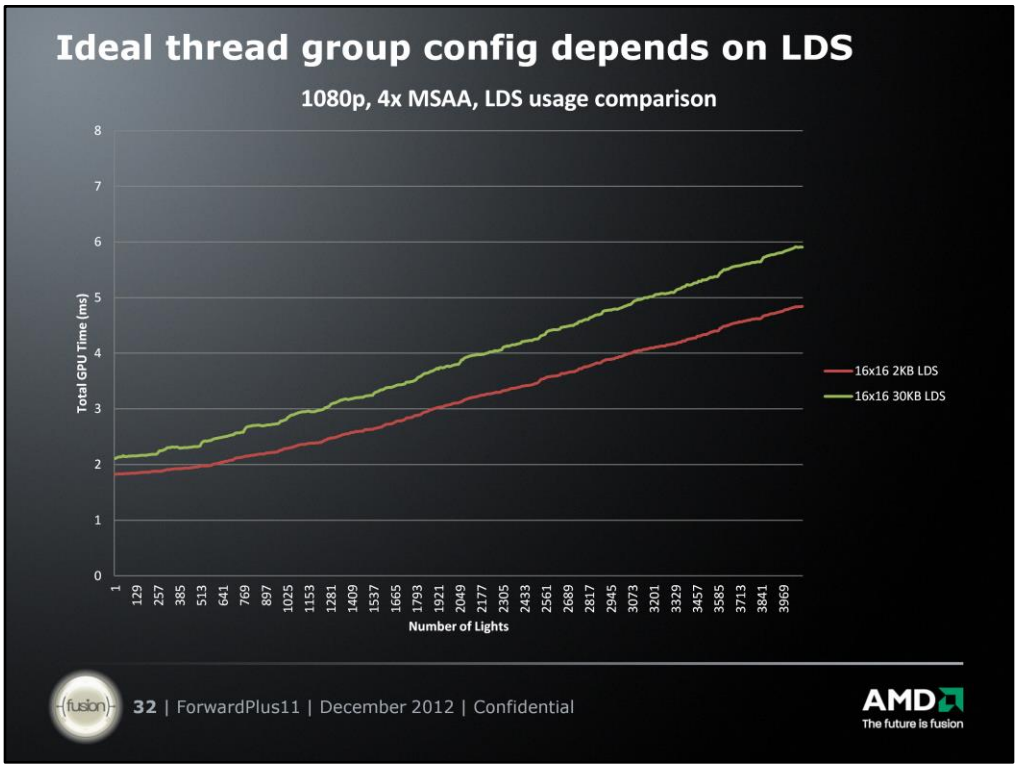


More threads per thread group is often better for performance. Remember the "multiple of 64" rule, tough. 9x9 tile size is more threads per thread group than 8x8, but is not a multiple of 64, so it's slower than 8x8.

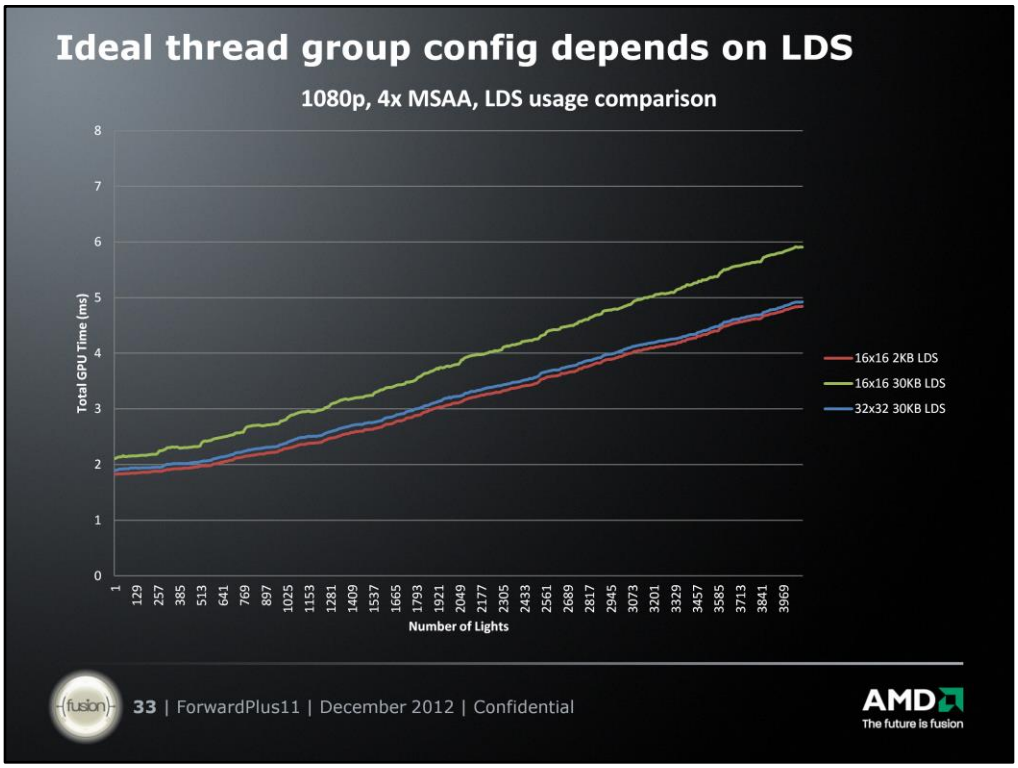
16x16 is a multiple of 64, and is much faster than 8x8 (at least for the particular case of the Forward+ algorithm).



Indeed, more threads per thread group is often better. But as you keep going to higher and higher thread counts, at some point it may no longer be faster. In the case of the Forward+ algorithm, a 32x32 tile size does speed up the compute shader execution time a little vs. 16x16. However, the larger tile size means you are culling at a coarser granularity, leading to more false positive light intersections, which slows down the forward pixel shader.



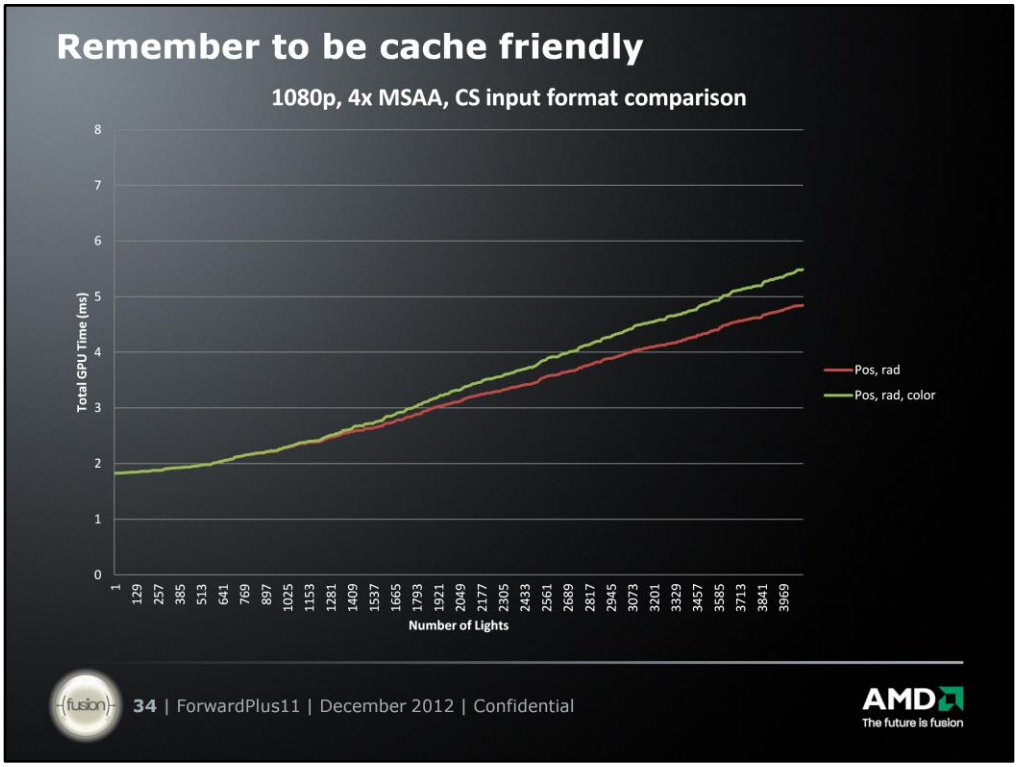
Here, the groupshared declaration for ldsLightIdx in ForwardPlus11Tiling.hlsl was changed to use 30KB. No other changes were made. Using more LDS (aka thread group shared memory, TGSM) than you need can result in scheduling inefficiencies in the Compute Units (CUs), leading to slower performance.



If the Forward+ algorithm actually needed 30KB, then using a 32x32 thread group configuration gets you most of the performance back (vs. a 16x16 using only 2KB LDS).

Two points here:

1. Use as little LDS as possible. Don't eat up 32KB just because the DX API allows it.
2. The "sweet spot" for your thread group config depends on LDS usage. There is no "one true answer" for the best thread group config. Experiment with your particular algorithm.



"Pos, rad" line is with light position and radius stored as a single float4. Light color is stored in a separate buffer.

```
Buffer<float4> g_LightBufferPositionAndRadius : register( t0 );
```

"Pos, rad, color" line stores the color as a float4 in addition to the position and radius float4, all in the same buffer.

```
struct LightArrayData
{
        float4 v4PositionAndRadius;
        float4 v4Color;
};
StructuredBuffer<LightArrayData> g_LightBuffer : register( t0 );
```

The compute shader only needs the position and radius. Storing addition light information in the same buffer means data is being brought into cache during CS execution that is never used, reducing cache efficiency.

The point here is twofold:

1. Don't use a float4 for color storage when a uint will do. Or more generally, use smaller storage formats where possible.
2. Organize your data to be cache friendly.

Summary

- Forward+ == Forward + Light Culling
- Use the power of Direct Compute to cull lights on the GPU
  - Solves the "lots of lights" problem for forward rendering
- Material variety
- MSAA is "automatic"
- Bandwidth friendly
- Supports transparency
- Straightforward implementation
- We have sample code

AMD
The future is fusion

Extensions

- Leo demo dynamically generates lights for GI that feed into the Forward+ algorithm
- Render scene from viewpoint of light
  - Write out info needed to spawn lights later
  - e.g. normal, albedo texture, depth, and then reconstruct position from depth
  - Think of it as a miniature G-Buffer

AMD The future is fusion

We plan to add this feature to the Forward+ sample in the future.

Extensions

- Leo demo dynamically generates lights for GI that feed into the Forward+ algorithm
- Use a Compute Shader to spawn lights
  - Each compute shader thread reads the light info from one pixel in the RT(s)
  - Then generates the info for a light, appending it to the light list that feeds into the Forward+ culling step

AMD The future is fusion

We plan to add this feature to the Forward+ sample in the future.

References

- Technology Behind AMD's "Leo Demo", Jay McKee
  - http://developer.amd.com/gpu_assets/AMD_Demos_LeoDemoGDC2012.ppsx
- Forward+: Bringing Deferred Lighting to the Next Level
  - http://diglib.eg.org/EG/DL/conf/EG2012/short/