



30 JULY – 3 AUGUST *Los Angeles*
SIGGRAPH2017

Future Directions for Compute-for-Graphics

Andrew Lauritzen
Senior Rendering Engineer, SEED
@AndrewLauritzen



Who am I?

- Senior Rendering Engineer at SEED – Electronic Arts
 - @SEED, <https://www.ea.com/SEED>
- Previously
 - Graphics Software Engineer at Intel (DX12, Vulkan, OpenCL 1.0, GPU architecture, R&D)
 - Developer at RapidMind (“GPGPU”, Cell SPU, x86, ray tracing)
- Notable research
 - Deferred shading (Beyond Programmable Shading 2010 and 2012)
 - Variance shadow maps, sample distribution shadow maps, etc.

Renderer Complexity is Increasing Rapidly

- Why?
 - Demand for continued increases in quality on wide variety of hardware
 - GPU fixed function not scaling as quickly as FLOPS
 - Balance of GPU resources is rarely optimal for a given pass, game, etc.
 - Power efficiency becoming a limiting factor in hardware/software design
- Opportunity to render “smarter”
 - Less brute force use of computation and memory bandwidth
 - Algorithms tailored specifically for renderer needs and content

Rendering Pipeline Overview

INSTANCE CULLING (FRUSTUM/OCCCLUSION)

CLUSTER CHUNK EXPANSION

CLUSTER CULLING
(FRUSTUM/OCCCLUSION/TRIANGLE BACKFACE)



INDEX BUFFER COMPACTION

MULTI-DRAW

SIGGRAPH 2015: Advances in Real-Time Rendering course

From "GPU-Driven Rendering Pipelines" by Ulrich Haar and Sebastian Aaltonen, SIGGRAPH 2015

BF1 : PS4™ Pro

1600x1800 ish...

Dynamic resolution scaling

3200x1800 ish...

3840x2160

- Clear (IDs and Depth)
- Partial Z-Pass
- G-Buffer Laydown
- Resolve AA Depth
- G-Buffer Decals
- HBAO + Shadows
- Tiled Lighting + SSS
- Emissive
- Sky
- Transparency
- Velocity Vectors

- CB Resolve + Temporal AA
- Motion Blur
- Foreground Transparency
- Gaussian Pyramid
- Final Post-Processing
- Silhouette Outlines

- Display Mapping + Resample



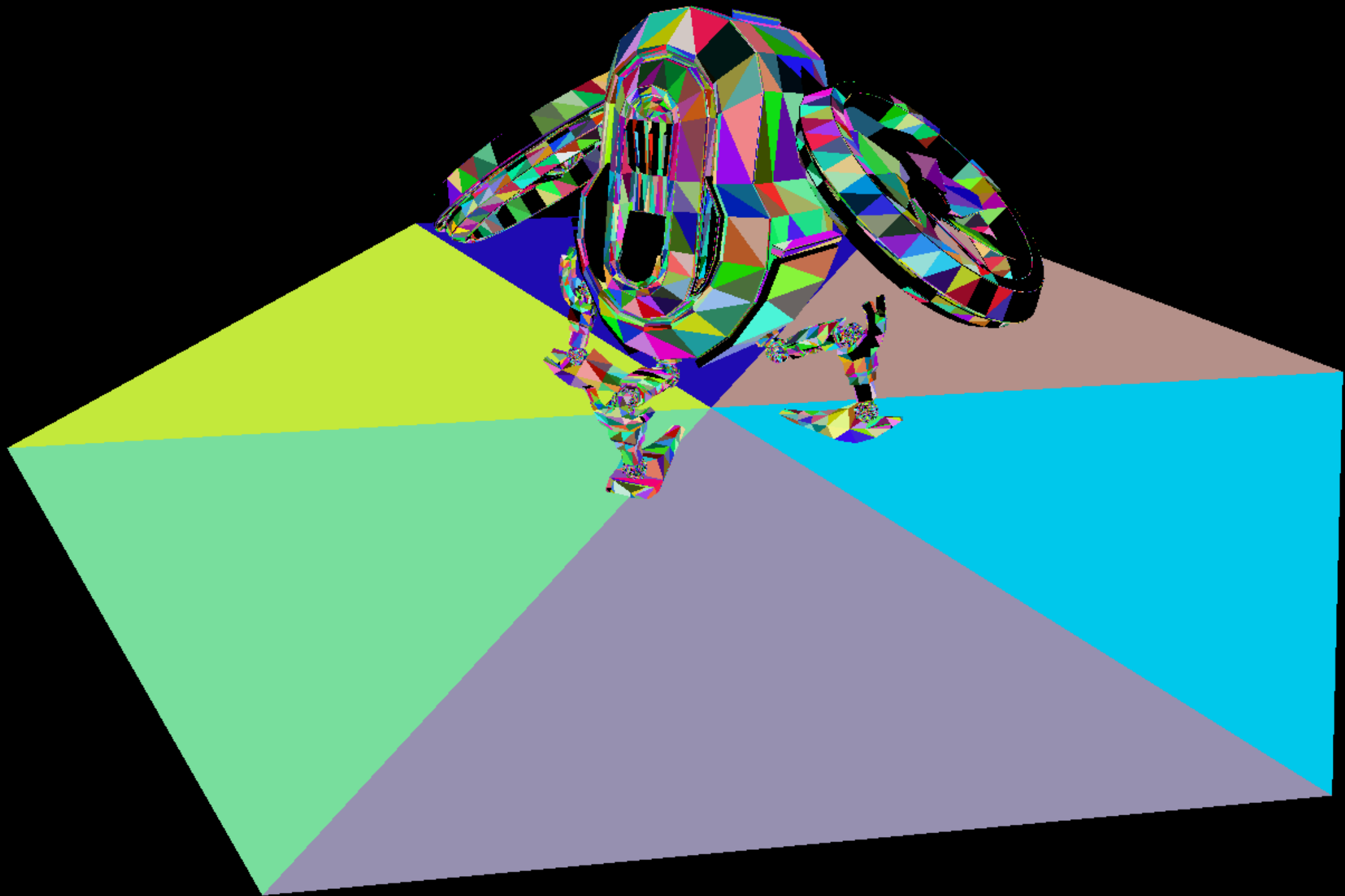
GPU Compute Ecosystem

- Complex algorithms and data structures are impractical to express
 - End up micro-optimizing the simple algorithms into local optima
- Significant language and hardware issues impede progress
 - Writing portable, composable *and* efficient GPU code is often impossible
- GPU compute for graphics has been mostly stagnant
 - Most of the problems I will discuss were known ~7 years ago! [BPS 2010]
 - CUDA and OpenCL have progressed somewhat, but not practical for game use

Case Study: Lighting and Shading

Lighting and Shading

- Deferred shading gives us more control over scheduling shading work
 - ... but still a chunk of it lives in the G-buffer pass
 - Per pixel state is large enough to inhibit multi-layer G-buffers
 - Want better control over shading rates, scheduling, etc.
- Visibility Buffers [Burns and Hunt 2013]
 - Follow-up work by Tomasz Stachowiak on GCN [Stachowiak 2015]
 - Store minimal outputs from rasterization, defer attribute interpolation
 - Possibly even defer full vertex shading
 - Use the rasterizer just to intersect primary rays, rest in compute



Visibility Buffer Shading

- We want to run different shaders based on material/instance ID
 - Similar need in deferred shading or ray tracing
- Sounds like dynamic dispatch via function pointer or virtual
 - Nice and coherent in screen space

```
struct Material {  
    Texture2D<float4> albedo;  
    float roughness;  
    ...  
    virtual float4 shade(...) const;  
};
```



Übershader Dispatch

```
struct Material
{
    int id;
    Texture2D<float4> albedo;
    float roughness;
    ...
};

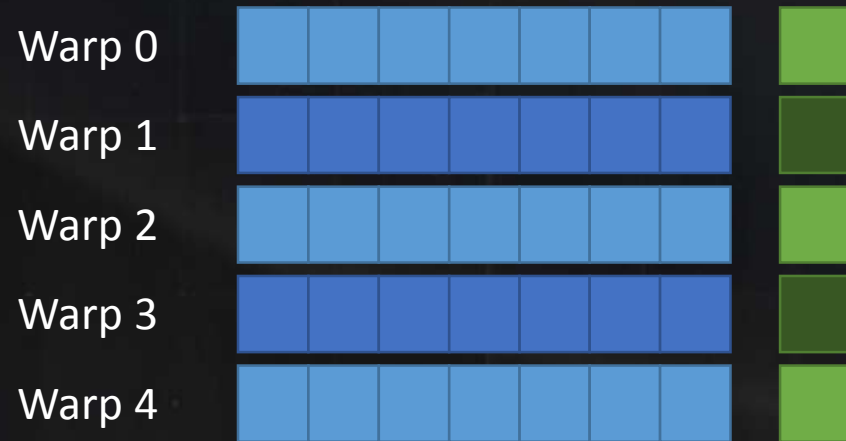
float4 shade(Material* material)
{
    switch (material->id)
    {
        case 0: return shaderA(material); // Entire function gets inlined
        case 1: return shaderB(material); // Entire function gets inlined
        case 2: return shaderC(material); // Entire function gets inlined
        ...
    };
}
```

Shader Resource Allocation

- Why are we forced to write code like this?
 - GPUs have been fully capable of dynamic jumps/branches for some time now
 - Why not just define a calling convention/ABI and be done with it?
- Static resource allocation
 - GPUs need to know how many resources a shader uses before launch
 - Can only launch a warp if all resources are available
 - Will tie up those resources for entire duration of the shader
 - Function calls either get inlined, or all potential targets must be known
- “Occupancy” is a key metric for getting good GPU performance
 - GPUs rely on switching between warps to hide latency (memory, instruction)

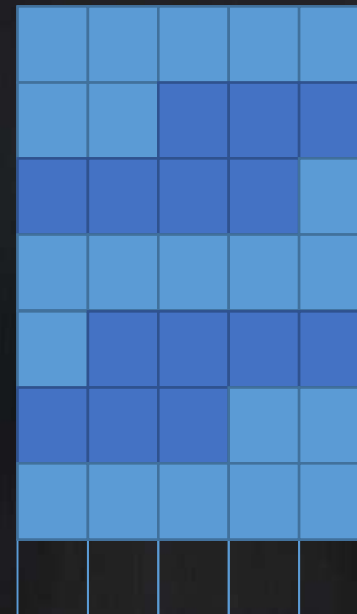
Shader Resource Allocation Example

```
return shaderA(material);
```



=> "Occupancy 5"

Registers (40)



Shared Memory (12)



Shader Resource Allocation Example

```
switch (material->id)
{
    case 0: return shaderA(material);
    case 1: return shaderB(material);
};
```

Warp 0



Warp 1



=> "Occupancy 2"

Registers (40)




Shared Memory (12)



Improving Occupancy via Software?

So how about **ten** dispatcher shaders?

- Check the SGPR, VGPR, and LDS usage of each material shader
- Bin it depending on max occupancy
 - Exactly ten bins 
- Create a shader for each bin
- Pre-sort materials by bin
- Shade everything with ten dispatches
 - Still just one sub-allocated pixel list
- Now shading at proper occupancy

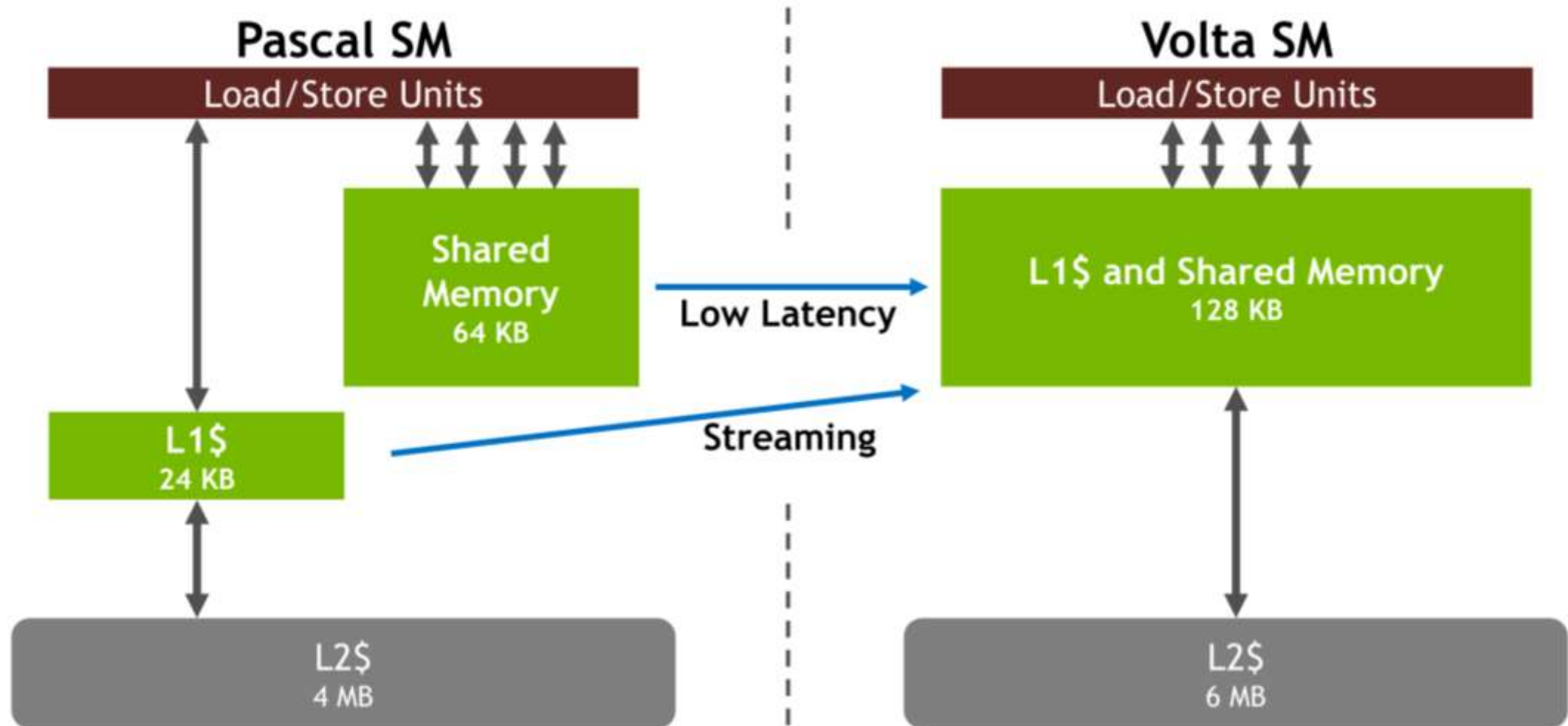
- Per IHV...
- Per architecture...
- Per SKU...
- Per driver/compiler...

From "A deferred material rendering system" by Tomasz Stachowiak, 2017

Improving Occupancy via Hardware

- Dynamic resource allocation/freeing inside warps?
 - Some potential here, but can get complicated quickly for the scheduler
- Compile everything for good occupancy
 - Let caches handle the dynamic variation via spill to L1\$?
- GPU caches are typically small relative to register files
 - Generally a poor idea to rely on them to help manage dynamic working set
 - ... but improvements are starting to emerge!

UNIFYING KEY TECHNOLOGIES



NARROWING THE SHARED MEMORY GAP

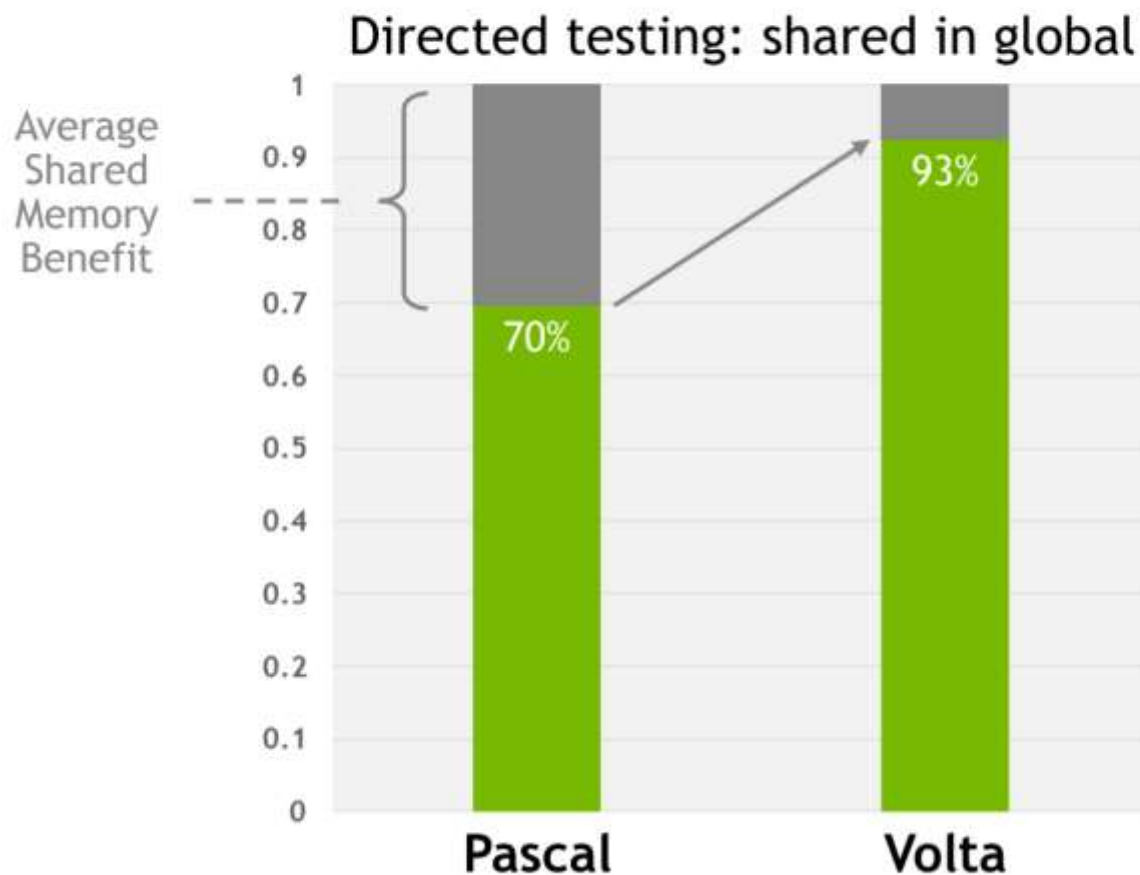
with the GV100 L1 cache

Cache: vs shared

- Easier to use
- 90%+ as good

Shared: vs cache

- Faster atomics
- More banks
- More predictable



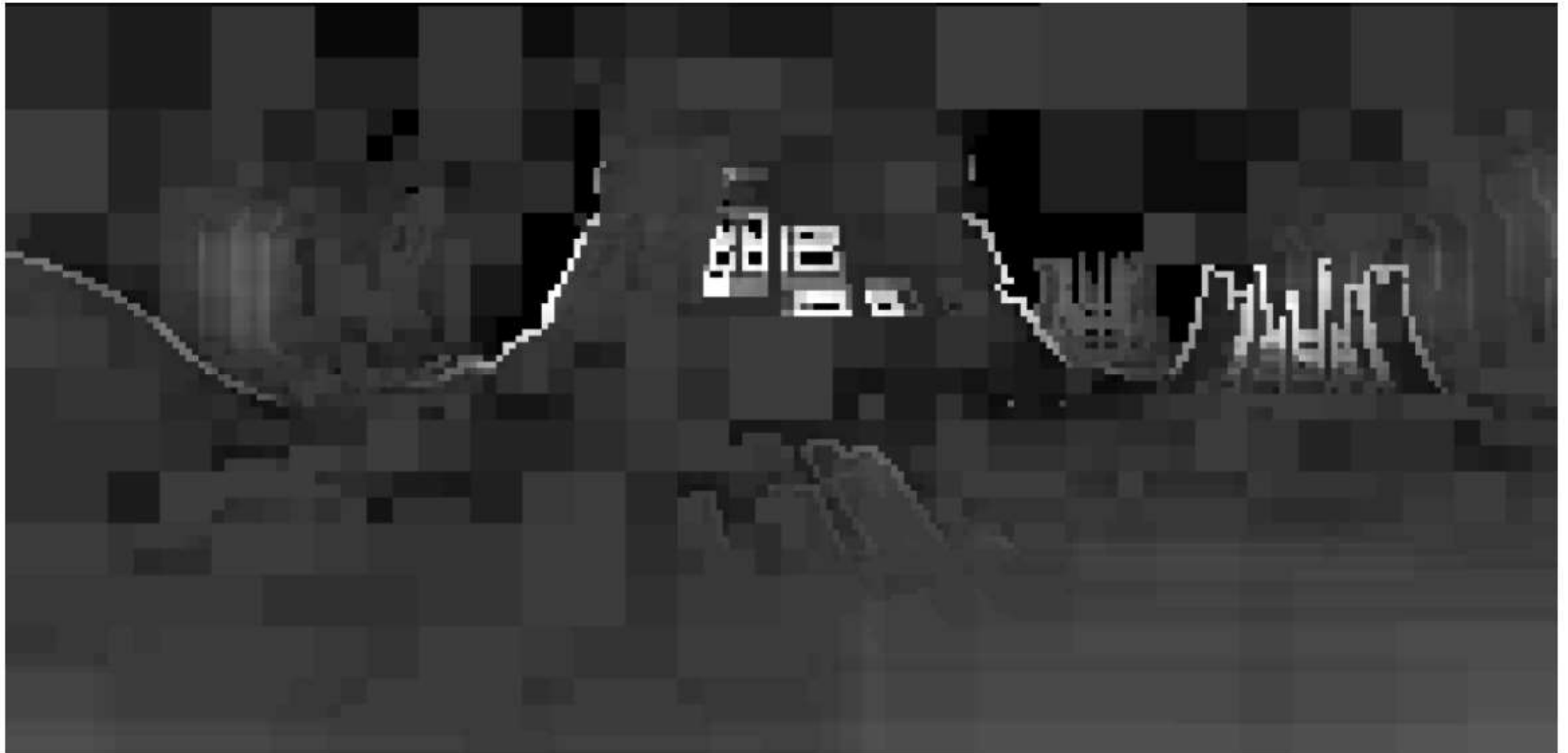
Visibility Buffer Takeaways

- Hardware needs improvements
 - More dynamic resource allocation and scheduling
 - Less sensitive to “worst case” statically reachable code
 - NVIDIA Volta appears to be a step in the right direction
- Enables software improvements
 - Dynamic dispatch
 - Separate compilation
 - Compose and reusable shader code

Case Study: Hierarchical Structures

Hierarchical Structures

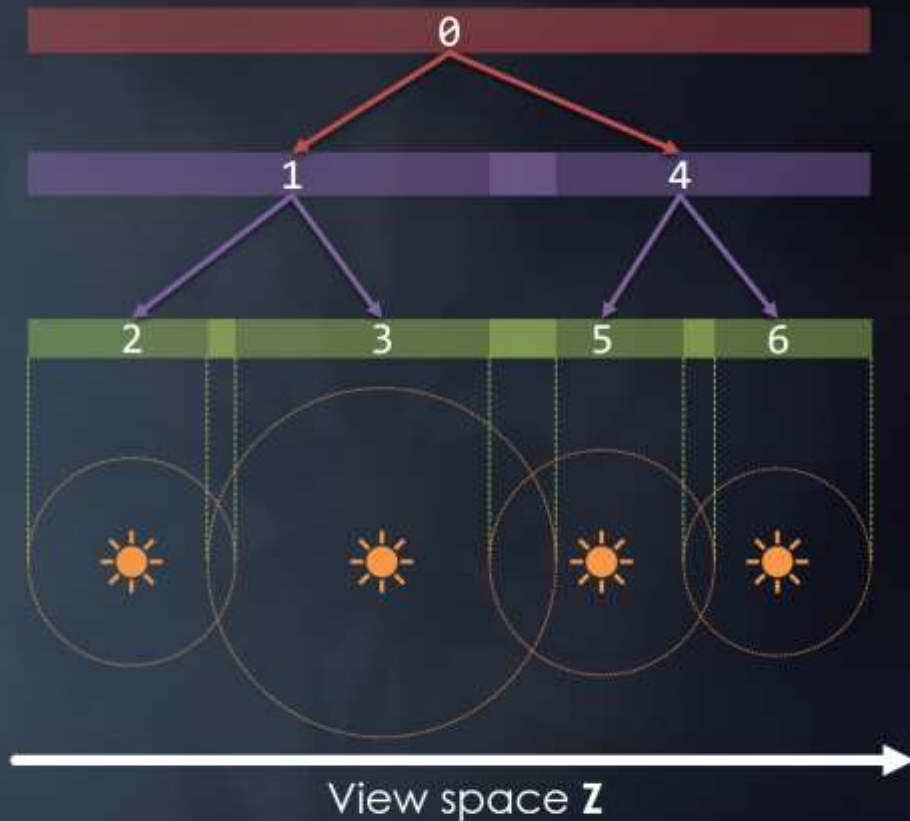
- Hierarchical acceleration structures are a good tool for “smarter” rendering
 - Mipmaps, quadtrees, octrees, bounding volume/interval hierarchies, ...
- Constructing these structures is inherently nested parallel
 - Data propagated bottom-up, top-down or both
 - Conventionally created on the CPU and consumed by the GPU
- Several reasons we want to construct these structures on the GPU as well
 - Latency: input data may be generated by rendering
 - FLOPS are significantly higher on the GPU on some platforms



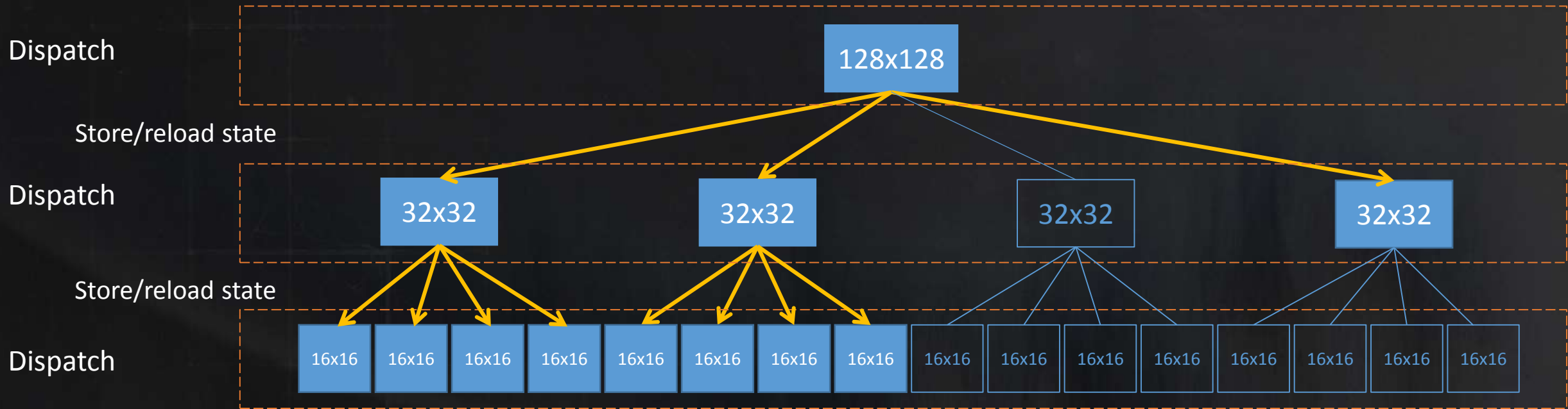
Light tree structure

- ▶ Bounding interval hierarchy over lights
 - ▶ **Complete binary tree**
 - ▶ 1D, using light **Z** extents
- ▶ Depth-first memory layout
- ▶ Lists of lights in leaf nodes

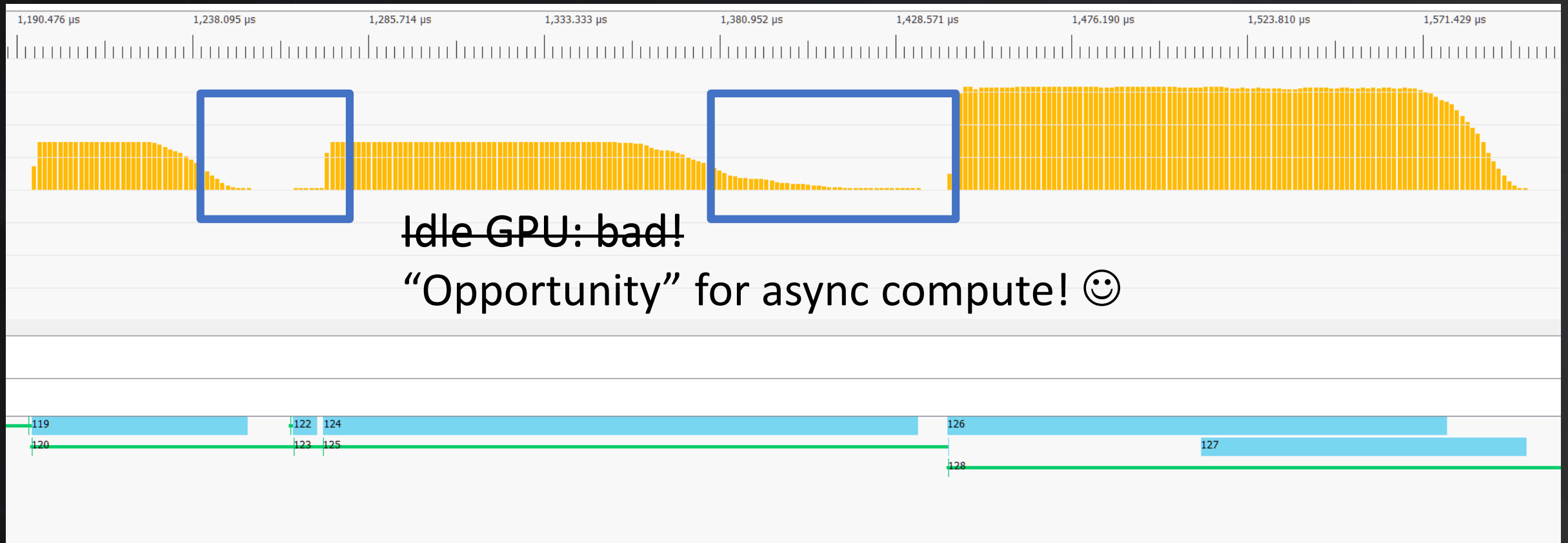
```
struct LightTreeNode
{
    float minDepth;
    float maxDepth;
    unsigned firstLightIndex;
    unsigned isLeaf : 1;
    unsigned lightCount : 16;
    unsigned skipCount : 15;
};
```



Quadtree GPU Dispatch



GPU Dispatch Inefficiency



From AMD's Radeon GPU Profiler

Quadtree GPU Block Reduction

```
#define BLOCK_SIZE 16
groupshared TileData tiles[BLOCK_SIZE][BLOCK_SIZE];
groupshared float4 output[BLOCK_SIZE][BLOCK_SIZE];

float4 TileFlat(uint2 tid, TileData initialData)
{
    bool alive = true;
    ...
    for (int level = MAX_LEVELS; level > 0; --level)
    {
        if (alive && all(tid.xy < (1 << level)))
        {
            ...
            if (isBaseCase(tileData))
            {
                output[tid] = baseCase(tileData);
                alive = false;
            }
            ...
        }
        GroupMemoryBarrierWithGroupSync();
    }
    return output[tid];
}
```

- Baked into kernel
- Optimal sizes are highly GPU dependent
- May not align with optimal sizes for data
- Potential occupancy issues
- Some algorithms need a full TileData stack

Quadtree GPU Block Reduction

```
#define BLOCK_SIZE 16
groupshared TileData tiles[BLOCK_SIZE][BLOCK_SIZE];
groupshared float4 output[BLOCK_SIZE][BLOCK_SIZE];

float4 TileFlat(uint2 tid, TileData initialData)
{
    bool alive = true;
    ...
    for (int level = MAX_LEVELS; level > 0; --level)
    {
        if (alive && all(tid.xy < (1 << level)))
        {
            ...
            if (isBaseCase(tileData))
            {
                output[tid] = baseCase(tileData);
                alive = false;
            }
            ...
        }
        GroupMemoryBarrierWithGroupSync();
    }
    return output[tid];
}
```

• Lots of GPU threads doing useless work


• Can't terminate early

• Can't re-pack threads

Quadtree GPU Block Reduction

```
#define BLOCK_SIZE 16
groupshared TileData tiles[BLOCK_SIZE][BLOCK_SIZE];
groupshared float4 output[BLOCK_SIZE][BLOCK_SIZE];

float4 TileFlat(uint2 tid, TileData initialData)
{
    bool alive = true;
    ...
    for (int level = MAX_LEVELS; level > 0; --level)
    {
        if (alive && all(tid.xy < (1 << level)))
        {
            ...
            if (isBaseCase(tileData))
            {
                output[tid] = baseCase(tileData);
                alive = false;
            }
            ...
        }
        GroupMemoryBarrierWithGroupSync();
    }
    return output[tid];
}
```

- Base and subdivide cases can't do thread sync
 - Can't (efficiently) use shared memory
 - Breaks nesting abstraction
- 

Improving Performance

Writing Fast Compute Shaders

The diagram illustrates the optimization of a compute shader by removing barriers and using register swizzling to bypass the warp width abstraction. The original code on the left is transformed into the optimized code on the right.

Original Code (Left):

```
groupshared uint localValidDraws;
[numthreads(1, 1, 1)]
void main()
{
    GroupMemoryBarrierWithGroupSync();

    MultiDrawIndirectArgs drawArgs;
    if (drawArgs.indexCount > 0)
        InterlockedAdd(localValidDraws, 1);

    GroupMemoryBarrierWithGroupSync();

    uint localSlot;
    if (drawArgs.indexCount > 0)
        storeIndirectDrawArgs(globalSlot + localSlot, drawArgs);
}
```

Optimized Code (Right):

```
[numthreads(64, 1, 1)] /** DO NOT CHANGE!! **/
void main(uint3 globalId : SV_DispatchThreadID,
          uint3 threadId : SV_GroupThreadID)
{
    const uint laneId = threadId.x;

    const uint drawArgId = globalId.x;
    const uint drawArgCount = batchData[g_batchIndex].drawCount;

    MultiDrawIndirectArgs drawArgs;
    if (drawArgId < drawArgCount)
        loadIndirectDrawArgs(drawArgId, drawArgs);

    const bool thisLaneActive = (drawArgs.indexCount > 0);
    uint2 clusterValidBallot = __XB_Ballot64(thisLaneActive);
    uint outputArgCount = __XB_S_BCNT1_U64(clusterValidBallot);
    uint localSlot = __XB_MBCNT64(clusterValidBallot);

    uint globalSlot;
    if (laneId == 0)
        InterlockedAdd(batchData[g_batchIndex].drawCountCompacted,
                       outputArgCount, globalSlot);

    globalSlot = __XB_ReadLane(globalSlot, 0);

    if (drawArgId < drawArgCount && thisLaneActive)
        storeIndirectDrawArgs(globalSlot + localSlot, drawArgs);
}
```

Annotations:

- Remove barriers (numthreads = hardware warp size)**: Points to the change in the `[numthreads]` macro from `(1, 1, 1)` to `(64, 1, 1)`.
- Remove smem and atomics Replace with register swizzling**: Points to the removal of `GroupMemoryBarrierWithGroupSync()` and `InterlockedAdd`, and the introduction of `__XB_Ballot64`, `__XB_S_BCNT1_U64`, `__XB_MBCNT64`, and `__XB_ReadLane`.
- i.e. bypass the warp width abstraction Write code directly for GPU warps**: Points to the overall transformation of the shader to be warp-specific.

From "Optimizing the Graphics Pipeline with Compute" by Graham Wihlidal, GDC 2016

Writing Fast *and Portable* Compute Shaders

- ... is just not possible right now in the compute languages we have
 - SIMD widths vary, sometimes even between kernels on the same hardware
 - Reliance on brittle compiler “optimization” steps to not fall off the fast path
 - Warp synchronous programming is not spec compliant [Perelygin 2017]
 - No legal way to “wait” for another thread outside your group [Foley 2013]
- Performance delta can be 10x or more
 - Too large to accept compatible “fallback”
- Is the SIMD width abstraction doing more harm than good?

More Explicit SIMD Model?

```
// No "numthreads": this is called once per warp always
void kernel(float *data)
{
    // Operations and memory at this scope are scalar, as you'd expect
    int x = 0;
    float4 array[128];

    // Does not have to match hardware SIMD size: compiler will add a loop in the kernel as needed
    parallel_foreach(int i: 0 .. N)
    {
        // This gets compiled to SIMD - this is like a regular shader in this scope
        float d = data[i] + data[i + N];

        // ALL the usual GPU per-lane control flow stuff works normally
        if (d > 0.0f)
            data[i] = d;

        // Parallel for's can be nested; various options on which scope/axis to compile to SIMD
        parallel_foreach(int j: 0 .. M) { ... }
    }

    // Another parallel/SIMD loop with different dimensions; no explicit sync needed
    parallel_foreach(int i: 0 .. X) { ... }
}
```

Quadtree Recursion on the GPU?

```
void TileRecurse(int level, TileData tileData)
{
    if (level == 0 || isBaseCase(tileData))
        return baseCase(tileData);
    else
    {
        // Subdivide and recurse
        TileData subtileData[4];
        computeSubtileData(level, tileData, subtileData);
        --level;

        spawn TileRecurse(level, subtileData[0]);
        spawn TileRecurse(level, subtileData[1]);
        spawn TileRecurse(level, subtileData[2]);
        spawn TileRecurse(level, subtileData[3]);
    }
}
```



*

error X3500: 'TileRecurse': recursive functions not allowed in cs_5_1

* exception: CUDA dynamic parallelism

CUDA Dynamic Parallelism

```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

From “How Tesla K20 Speeds Quicksort, a Familiar Comp-Sci Code”, Blog Post, [Link](#)

CUDA Dynamic Parallelism

- CUDA *does* support recursion including fork *and* join!
 - CUDA also supports separate compilation
- Performance issues remain
 - Mostly syntactic sugar for spilling, indirect dispatch, reload
 - Not really targeted at games and real-time rendering
- Definitely an improvement on the semantic front
 - Get similar capabilities into HLSL and implementations improve over time?
 - Chicken and egg problem with game performance (ex. geometry shaders)

CUDA Cooperative Groups

Flexible, Explicit Synchronization

Thread groups are explicit objects in your program

```
thread_group block = this_thread_block();
```

You can synchronize threads in a group

```
block.sync();
```

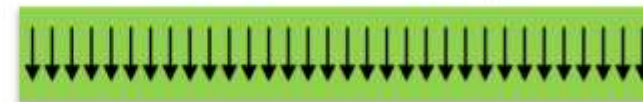
Create new groups by partitioning existing groups

```
thread_group tile32 = tiled_partition(block, 32);  
thread_group tile4 = tiled_partition(tile32, 4);
```

Partitioned groups can also synchronize

```
tile4.sync();
```

Thread Block Group



Partitioned Thread Groups



Hierarchical Structures Takeaways

- Nested parallelism and trees are key components of many algorithms
 - We need an efficient, portable way to express these
- Current compute languages are not particularly suitable
 - Only way to write efficient code is to defeat the abstractions
- We need new languages and language features
 - Map/reduce? Fork/join?
 - CUDA cooperative groups?
 - ISPC [Pharr 2012]? BSGP [TOG 2008]? Halide [Ragan-Kelly 2013]?
- Good opportunity for academia and industry collaboration!

Code Compatibility and Reuse

Compute Language *Basics*

- C-like, unless there's a compelling reason to deviate
 - Including all the regular C types... 8/16/32/64-bit
- Buffers
 - Structured buffers
 - “Byte address buffers” (really dword + two extra 0 bits for fun!)
 - Constant buffers, texture buffers, typed buffers, ...
- REAL pointers please!
 - We've already taken on all of the “negative” aspects with root buffers (DX12)
 - Now give us the positives

Compute Language *Basics*

- Resource binding and shader interface mess
 - Even just between DX12 and Vulkan this is a giant headache
- Get rid of “signature” and “layout” glue
 - Replace with regular structures and positional binding (i.e. “function calls”)
 - Pointers for buffers, bindless for textures
- DX12 global heap is an acceptable solution for bindless
 - Must be able to store references to textures in our own data structures
 - Ideal would be to standardize a descriptor size (or even format!)

Compute Language *Basics*

- Avoid/remove weird features that end up as legacy pain
 - UAV counters
 - Be very careful with shader “extensions”
- Mature JIT compilation setup
 - DXIL/SPIR-V make this both better and worse...
 - Shader compiler/driver cannot be allowed to emit compilation errors (!!)
 - We cannot have random errors on end user machines

Compute Language *Essentials*

- CPU/GPU interoperation and synchronization
 - Low latency, bi-directional submission, signaling and atomics
 - ... in *user space*
- Shared data structures
 - Consistent structure layouts and alignment between CPU/GPU

Call to Action

Compute for Graphics



Call to Action

- Hardware vendors
 - More dynamic resource allocation and scheduling
 - Less sensitive to “worst case” statically reachable code
 - Dynamic warp sorting and repacking
 - Unified compression and format conversions on all memory paths
- Operating system vendors
 - GPUs as first class citizens (like another “core” in the system)
 - Fast, user-space dispatch and signaling
 - Shared virtual memory

Call to Action

- Standards bodies
 - Be willing to make breaking changes to compute execution models
 - Forgo some low level performance (optionally) for better algorithmic expression
 - Resist being bullied by hardware vendors 😊
- Academics and language designers
 - Innovate on new efficient, composable, and robust parallel languages
 - Work with industry on requirements and pain points
- Game developers
 - Let people know that the status quo is *not* fine
 - Help the people who are working to improve it!

Conclusion

- Recently industry did explicit graphics APIs
 - Data-parallel intermediate languages (DXIL, SPIR-V)
- Next problem to solve is compute
 - Want to take real-time graphics further
 - Ex. 8k @ 90Hz stereo VR with high quality lighting, shading and anti-aliasing
 - Need to render “smarter”
- Requires industry-wide cooperation!

Acknowledgements

- Aaron Lefohn (@aaronlefohn)
- Alex Fry (@TheFryster)
- Colin Barré Brisebois (@ZigguratVertigo)
- Dave Oldcorn
- Graham Wihlidal (@gwhlidal)
- Jasper Bekkers (@JasperBekkers)
- Jefferson Montgomery (@jdmo3)
- Johan Andersson (@repi)
- Matt Pharr (@mattpharr)
- Natalya Tatarchuk (@mirror2mask)
- Neil Henning (@sheredom)
- Tim Foley (@TangentVector)
- Timothy Lottes (@TimothyLottes)
- Tomasz Stachowiak (@h3r2tic)
- Yuriy O'Donnell (@YuriyODonnell)

References

- Brodman, James et al. *Writing Scalable SIMD Programs with ISPC*, WPMVP 2014, [Link](#) [Web Site](#)
- Burns, Christopher and Hunt, Warren, *The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading*, JCGT 2013, [Link](#)
- Foley, Tim, *A Digression on Divergence*, *Blog Post in 2013*, [Link](#)
- Giroux, Olivier and Durant, Luke, *Inside Volta*, GTC 2017, [Link](#)
- Haar, Ulrich and Aaltonen, Sebastian, *GPU-Driven Rendering Pipelines*, *Advances in Real-Time Rendering 2015*, [Link](#)
- Harris, Mark, *CUDA 9 and Beyond*, GTC 2017, [Link](#)
- Hou, Qiming et al. *BSGP: Bulk-Synchronous GPU Programming*, TOG 2008, [Link](#)
- Jones, Stephen, *Introduction to Dynamic Parallelism*, GTC 2012, [Link](#)
- O'Donnell, Yuriy and Chajdas, Matthäus, *Tiled Light Trees*, I3D 2017, [Link](#)
- Perelygin, Kyrylo and Lin, Yuan, *Cooperative Groups*, GTC 2017, [Link](#)
- Pharr, Matt and Mark, William, *ISPC: A SPMD Compiler for High-Performance SIMD Programming*, InPar 2012, [Link](#)
- Ragan-Kelly, Jonathan et al. *Halide: A Language and Compiler for Optimizing Parallelism, Locality and Recomputation in Image Processing Pipelines*, PLDI 2013, [Link](#) [Web Site](#)
- Stachowiak, Tomasz, *A Deferred Material Rendering System*, *Blog Post 2015*, [Link](#)
- Wihlidal, Graham, *4K Checkerboard in Battlefield 1 and Mass Effect Andromeda*, GDC 2017, [Link](#)
- Wihlidal, Graham, *Optimizing the Graphics Pipeline with Compute*, GDC 2016, [Link](#)