Vulkanised 2026
The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

Mobile Ray Tracing Demystified: Techniques and Performance Tips

Iago Calvo Lista, Arm

arm

Mobile Ray Tracing Demystified: Techniques and Performance Tips

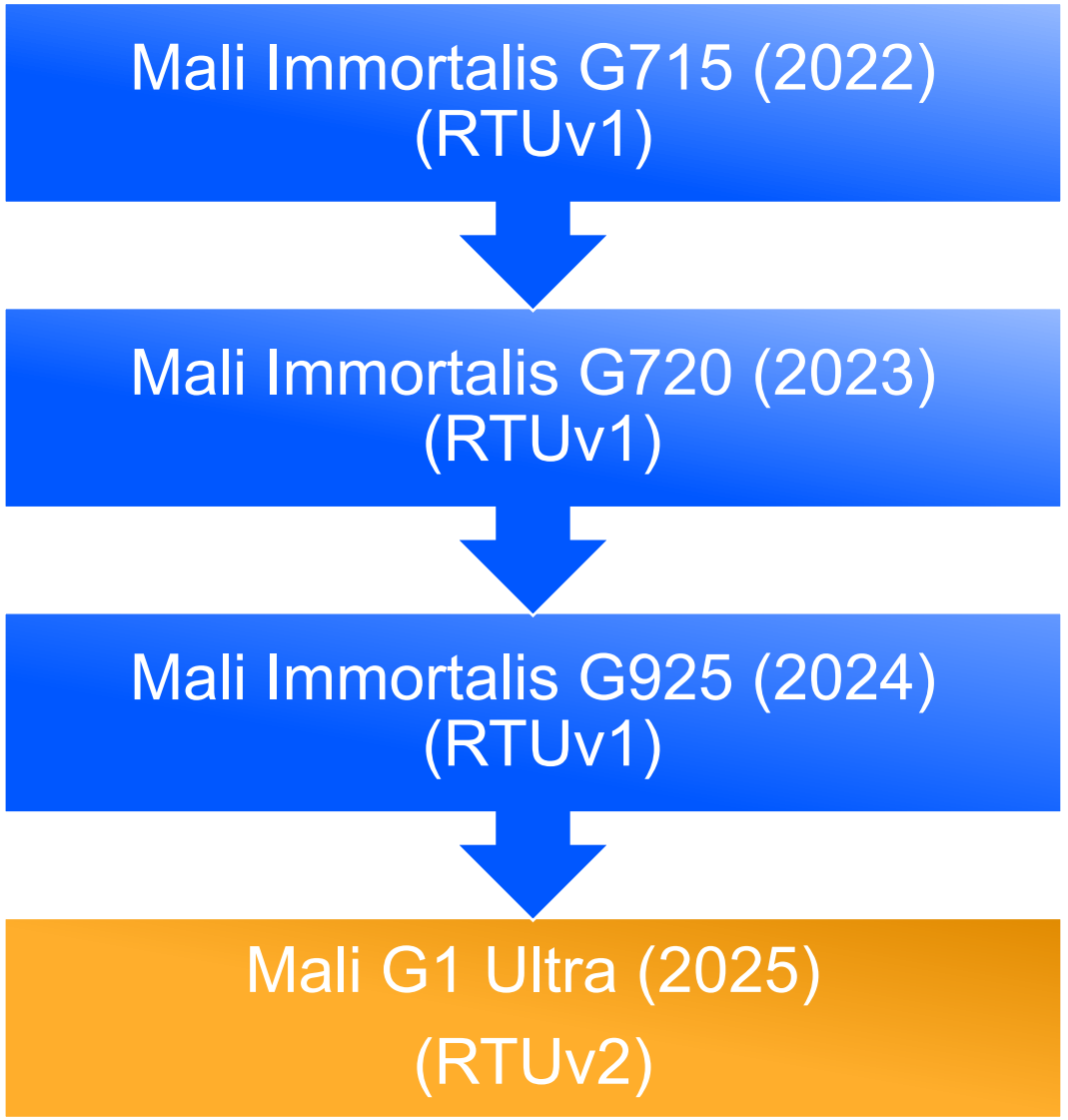
1. Introduction: Ray tracing on mobile
2. Vulkan Ray Tracing
    1. Acceleration Structures
    2. Ray Query and Ray Tracing Pipeline
3. Reflections
4. Reflections: Optimizations
    1. Hybrid (SSR+RQ)
    2. RQ Binning
5. Reflections: Solving the Material
    1. Ray Tracing Pipeline
    2. RQ Uber Shader
    3. RQ Visibility Buffer
6. Conclusions

arm

Introduction:
Ray Tracing on mobile

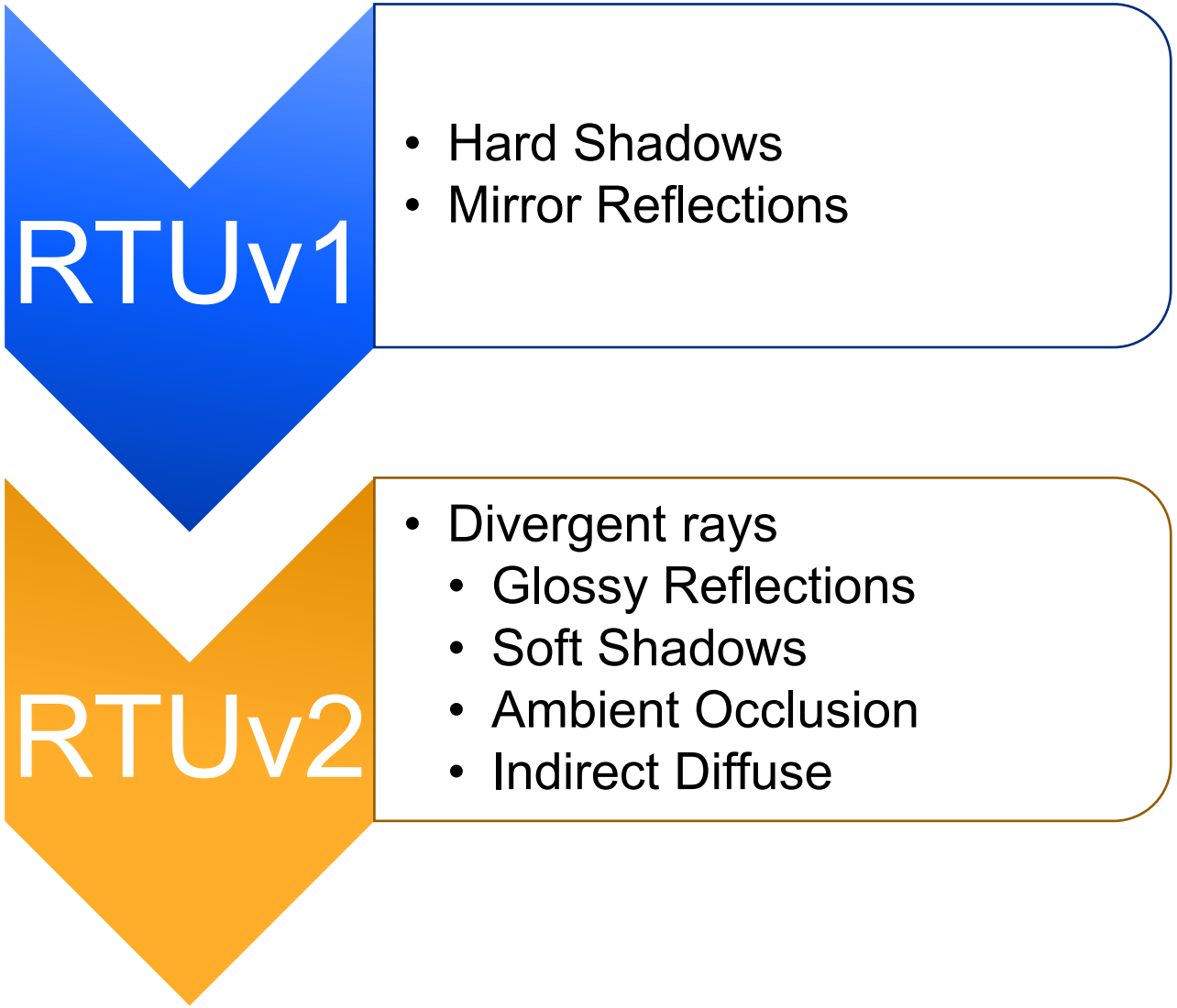
Ray Tracing on mobile

- Why should developers care about ray tracing on mobile?
- RT performances keeps improving
- Mali G1 Ultra:
  - New Ray Tracing Unit (RTUv2)
  - Main improvements with high ray divergence
    - Single ray model
  - Significant performance leap
    - Full hardware traversal
- Fully transparent upgrade for developers:
  - No new Vulkan extension
  - No code changes needed
  - Most existing best practices remain valid
- Ray tracing → visual quality improvement
  - Key differentiator for mobile games



RTUv2 - Performance

- Substantial performance gain
  - Accelerated ray traversal
  - Improves ray divergence
- Enables new techniques
- Improvements in AS build size
  - Not HW specific
- Good opportunity to re-evaluate performance
  - Performance characteristics changed
  - New techniques are now practical
  - Ecosystem adoption has increased significantly
  - Previous testing needs re-evaluation



Sponza – RT

AS: 353K triangles Resolution: 2300x1082 (shadows 1150x541 reflections 1150x1082)

Bistro – RT

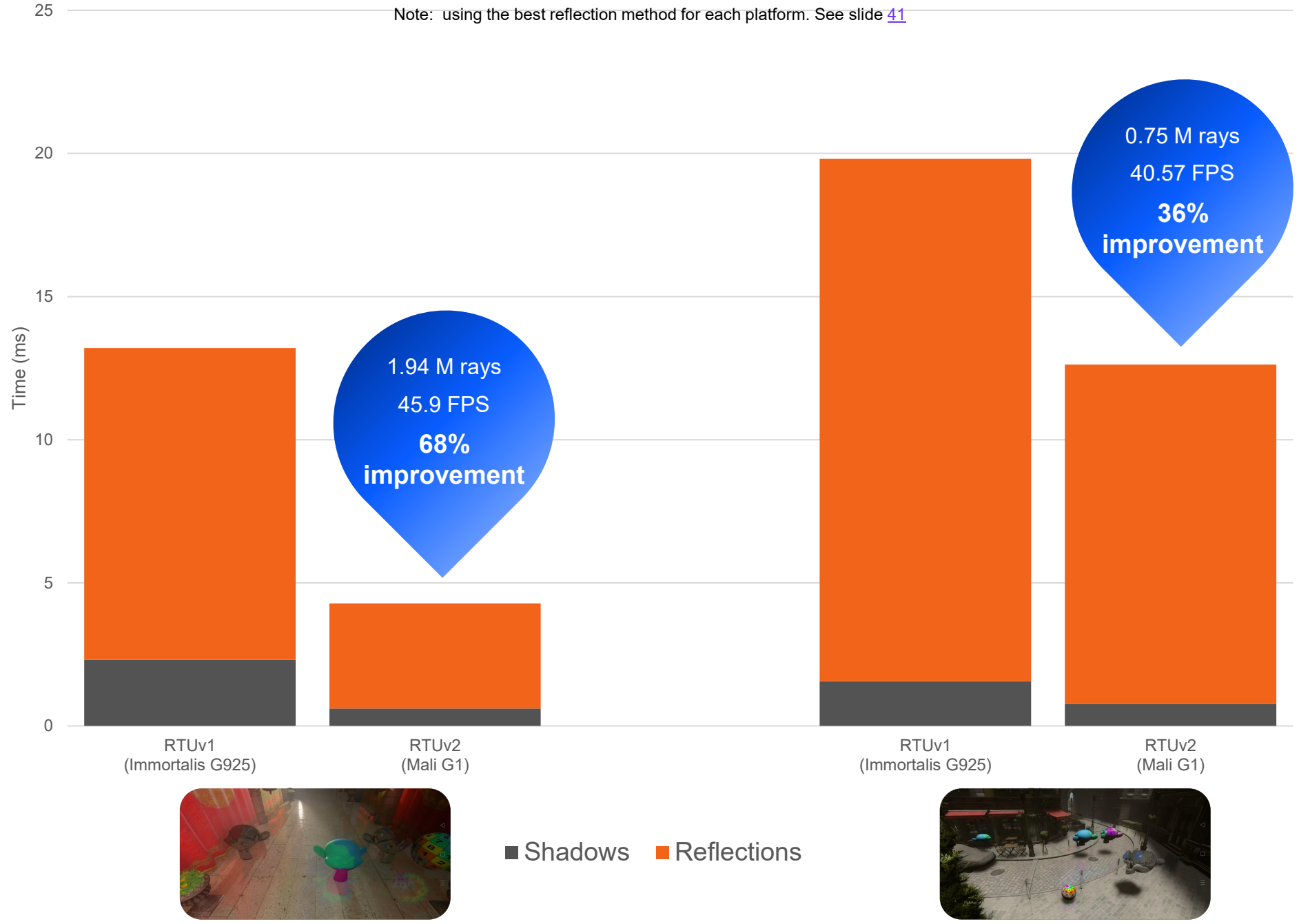
AS: 2.935M triangles Resolution: 1604x756 (shadows 802x378 reflections 802x756)

Profiling with Streamline

- Arm Streamline
- Part of Arm Performance Studio
- Support for Mali timeline
  - Per pass data
  - Vulkan labels

https://developer.arm.com/documentation/101816/0904/Analyze-your-capture/Viewing-application-activity/Mali-Timeline-mode

Note: using the best reflection method for each platform. See slide 41



arm

Vulkan Ray Tracing

1. Vulkan Ray Tracing API
2. Acceleration Structures
3. Ray Query and Ray tracing pipeline

How Vulkan Ray Tracing Works

- Ray tracing on Vulkan
  - More detail on previous talks
- This talk
  - Only basic concepts
  - Practical implementation tips

youtu.be/K19LttE67uQ?si=SukEIXmGiXZKqVEj

youtu.be/jJyHzkWXEfY?si=1XjOuTeqfkyYETsX

docs.vulkan.org/tutorial/latest/courses/18_Ray_tracing/00_Overview.html

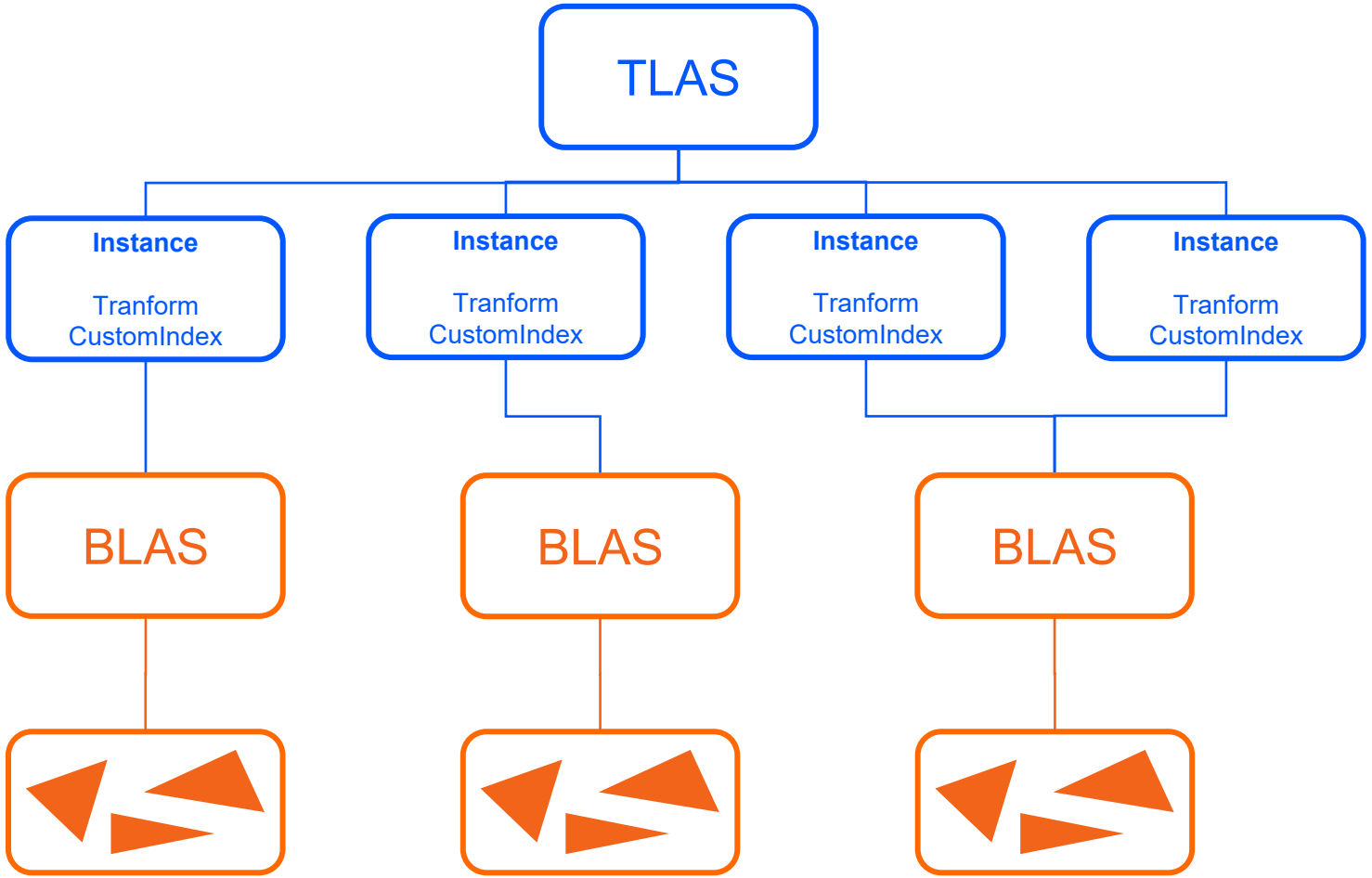
arm

Acceleration Structures

Vulkan Ray Tracing

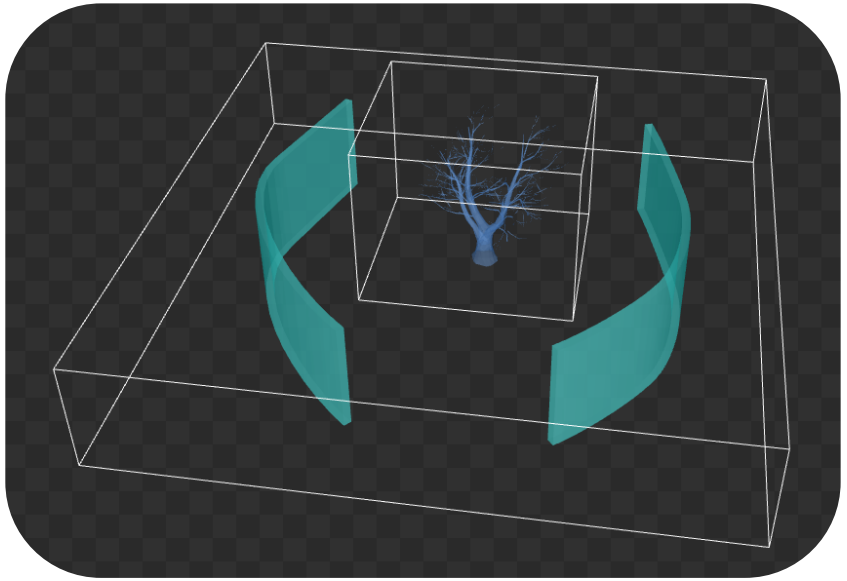
Acceleration Structures

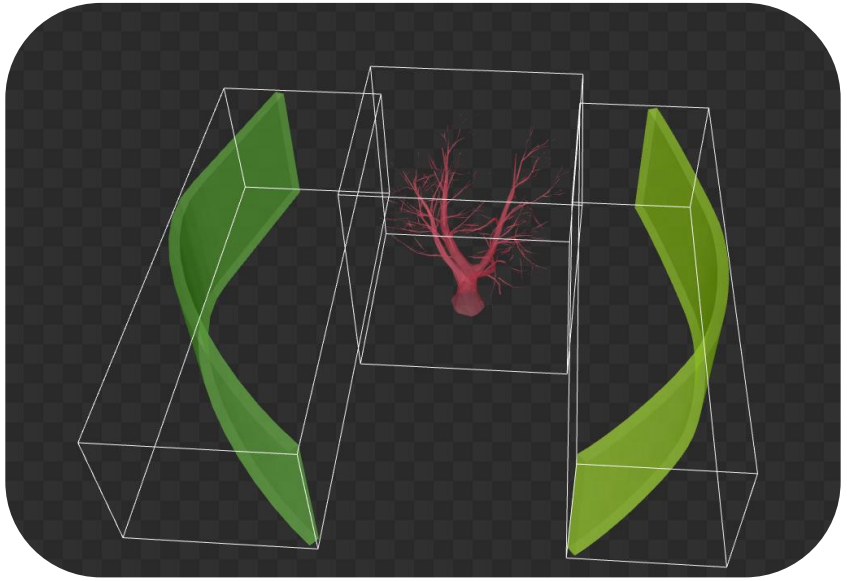
- Acceleration Structures (AS)
- Hierarchical data structures
- Organise geometry
- Usually Bounding Volume Hierarchy (BVH)
- Two types:
  - **Bottom Level AS (BLAS):**
    - Mesh vertex and index data
  - **Top Level AS (TLAS):**
    - Instances of BLAS with:
      - Transform Matrix: `transform`
      - Custom Id: `instanceCustomIndex`



Building the AS - Best practices

- Prefer `GEOMETRY_TYPE_TRIANGLES`
  - `GEOMETRY_TYPE_AABBS` still HW accelerated
- Avoid BLAS overlap, empty space and small BLASes:
  - Bigger `vkCmdBuildAccelerationStructures`
  - Improved on new drivers → Still recommended on most GPUs
- Use appropriate flags: Important for Mali
  - `FAST_TRACE`: Static BLASes
  - `FAST_BUILD`: TLASes and Skinned meshes
- Avoid full rebuild: Refit and update
- Mark opaque geometry
  - Try to use shader flags
    - Newer Mali Drivers:
      - `gl_RayFlagsSkipAABBEXT | gl_RayFlagsOpaqueEXT`
    - All Mali drivers:
      - `gl_RayFlagsSkipAABBEXT | gl_RayFlagsCullNoOpaqueEXT`
  - Mark opaque instances and geometries
- `VK_EXT_opacity_micromap`
  - Software implementation in current Mali GPUs
  - Developer experimentation → Early for performance conclusions





arm

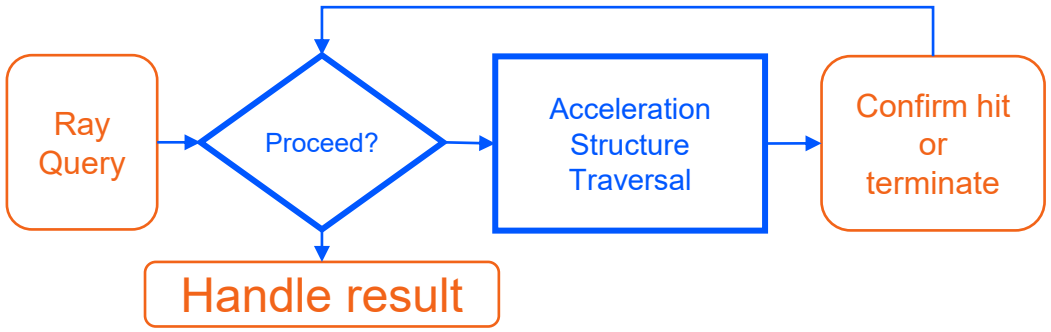
arm

Ray Query vs Ray Tracing Pipeline

Vulkan Ray Tracing

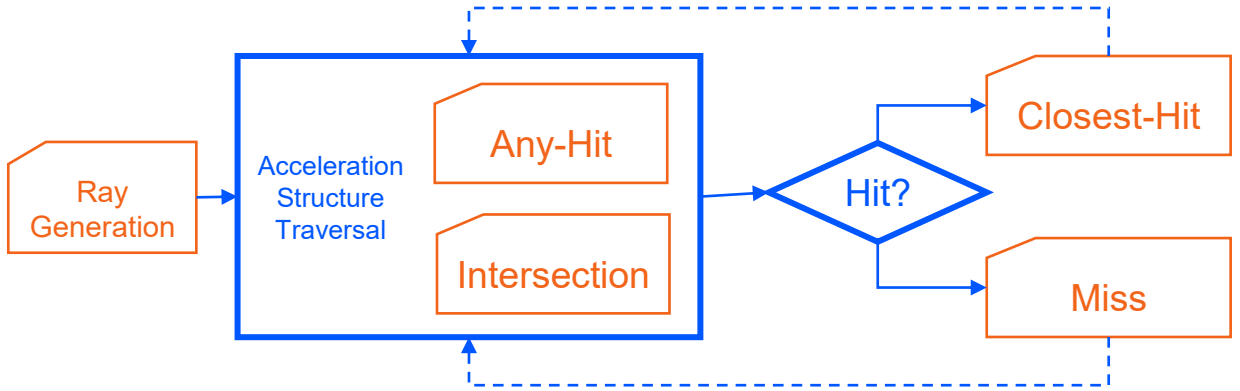
Ray Query

- Inline ray tracing
- Any shader stage
  - Easy to integrate
- Check for slow path in Mali
  - Mali offline compiler (**malioc**)
    - `Has slow ray traversal: false`
- All rays in a warp use the same TLAS.



Ray Tracing Pipeline

- Driver managed approach
- New shader stages
- Different use cases than RQ:
  - RQ: low number of hit shaders (AO, Shadows)
- Shader flags: Important for RTP and RQ
  - `gl_RayFlagsTerminateOnFirstHitEXT`:
    - 42.6% improvement on Bistro shadows (Mali G1)



Ray tracing pipeline – Mali best practices

- Very few mobile devices expose RTP
- RTP is quite slow on RTUv1 devices
  - Eg RQ inside hit shaders will trigger the S/W RT emulation
- RTP better fit for RTUv2 hardware
- Upcoming drivers for RTUv2 will include
  - Link-time optimizations of RTP pipelines (LTO)
  - Register assisted payload allocation
  - Many more improvements
- How to use LTO better:
  - Combines ray-gen + hit shaders into a big shader blob
  - LTO triggers when number of shader is low
  - If too many shaders, then it falls back to “function calls”
  - 10 shaders can be LTO-ed (including ray generation)
    - Maximum of 25 inline per pipeline (recursion)
    - May change in future drivers
  - Disabled when `VK_KHR_pipeline_library`
- General RTP tips:
  - Keep the ray payloads as small as possible

Keep ray payload small

Reduce total number of shaders

Avoid pipeline libraries

arm

Reflections

No Reflections

Glossy reflections

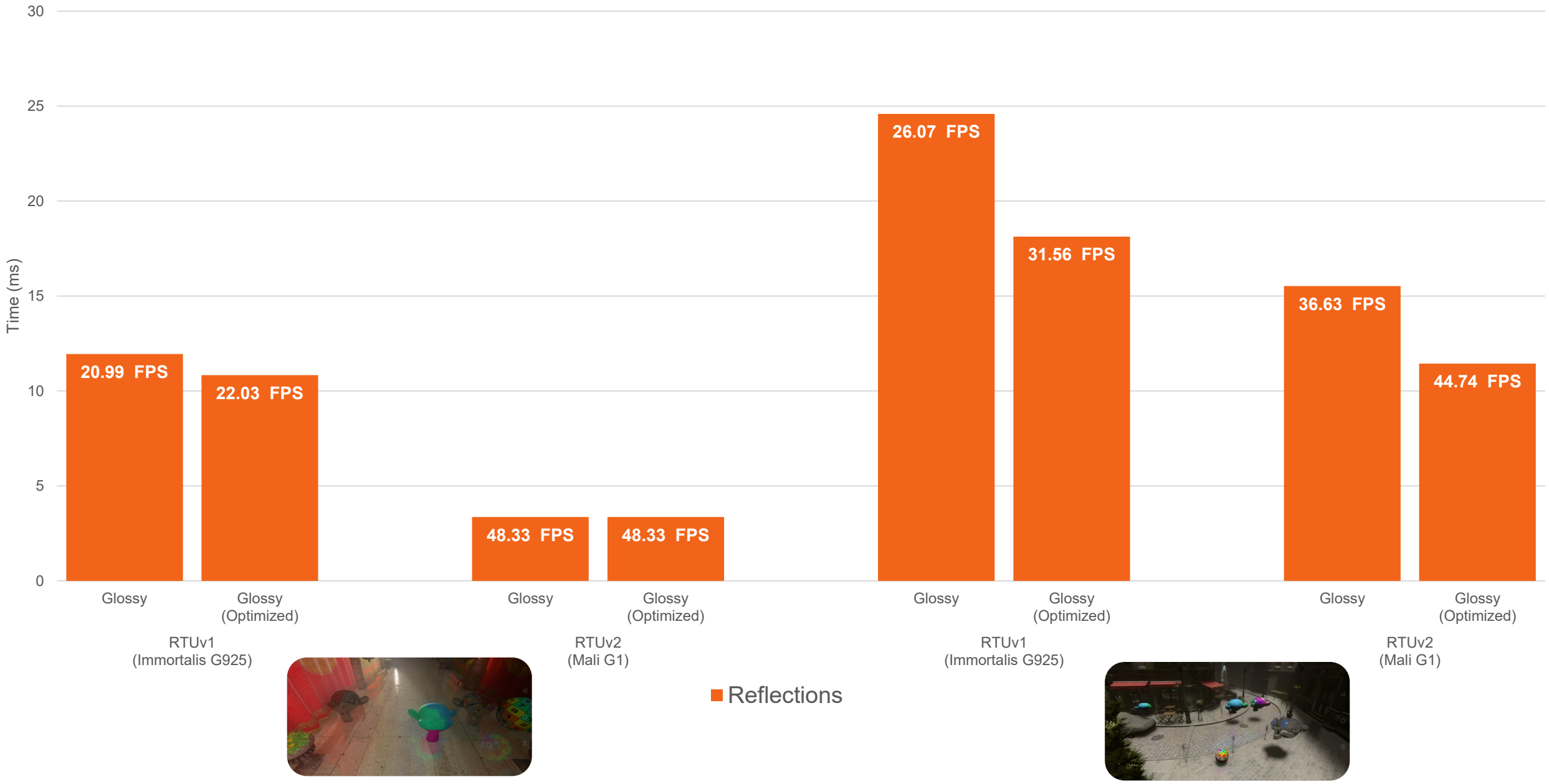
- Mirror reflections → unrealistic on rough surfaces
  - Glossy reflections → accurate representation
  - Subtle effect → Difficult to notice outside game
- Feasible in newer devices

Ray traced reflections

- Retrieve G-buffer data
- Check if we need RQ ray
- Compute reflection direction.
  - For glossy: Stochastically perturb direction.
- Launch RQ ray.
  - Hit: retrieve material data and evaluate lighting
    - Second RQ ray for direct illumination
    - Resolve material using `instanceCustomIndex`
  - Miss:
    - Skybox/environment map

```
getGbufferData(svDispatchThreadId.xy, ...);
if(isSky(depth))
{
  outputColor = Vec4(0); //No reflection in Sky
  return;
}
if(gbufferRoughness <= MIRROR_ROUGHNESS)
{
  reflDir = reflect(viewDir, gbufferNormal);
}
if(ALLOW_GLOSSY && gbufferRoughness <= MAX_RT_ROUGHNESS)
{
  reflDir = computeStochasticReflectionDir(...);
}
else
{
  outputColor = reflectionFallback();
  return;
}
traceRqReflectionRay(refDir, hitData, ...);
if(hitData.hit)
{
  bool directLightRq = traceShadowRay(hitData, ...);
  outputColor = sampleReflectionColor(
    hitData, directLightRq, ...); //use instanceCustomIndex
}
else
{
  outputColor = reflectionSky(...);
}
```

arm



Note: Optimized Reflections are using the best method for each platform. See slide 41

arm

Reflections Optimizations

1. Hybrid (SSR+RQ)
2. Ray binning

arm

Hybrid (SSR+RQ)

Reflections Optimizations

Hybrid (SSR+RQ) implementation

- PASS 1: SSR

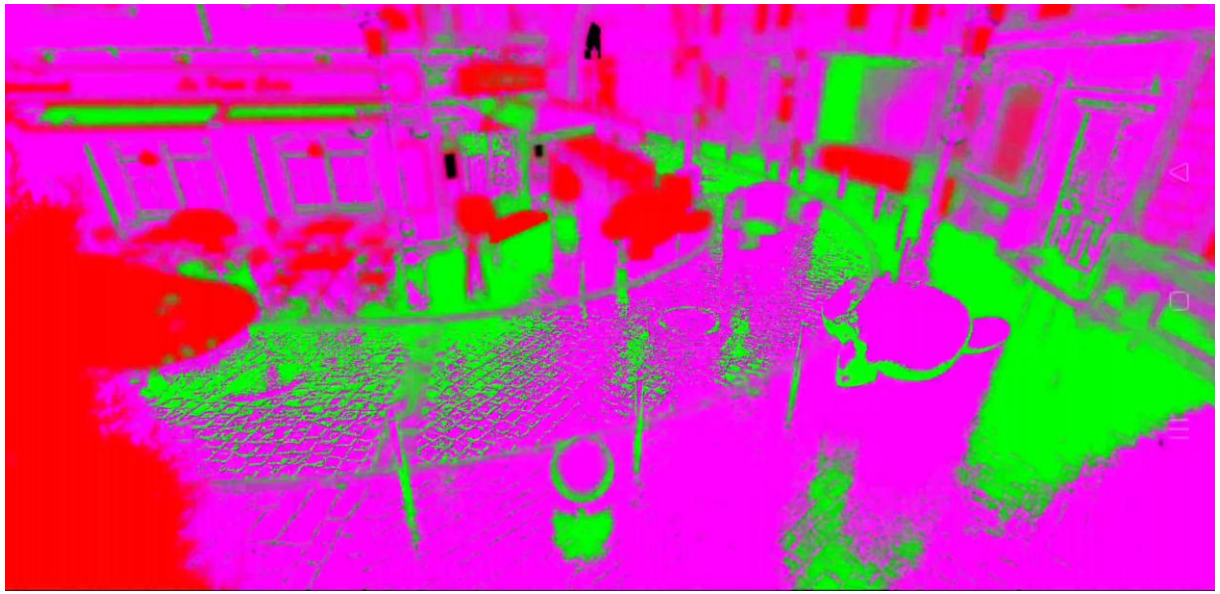
1. Retrieve data from Gbuffer
   - Stochastic ray direction.
2. Screen-space march against depth buffer.
   - Screen-space is cheap →Reduces RQ rays
3. Handle SSR result
   1. SSR Hit: resolve via G-buffer.
   2. SSR Miss: push pixel into RQ input buffer.
      - Origin: last visited SSR position
        - Reduce unnecessary checks.
      - Pixel allocation: handled via atomics.
   - Few RQ rays per subgroup
     - RQ ray directly → Low GPU utilisation
     - Compact RQ rays in buffer and launch indirectly

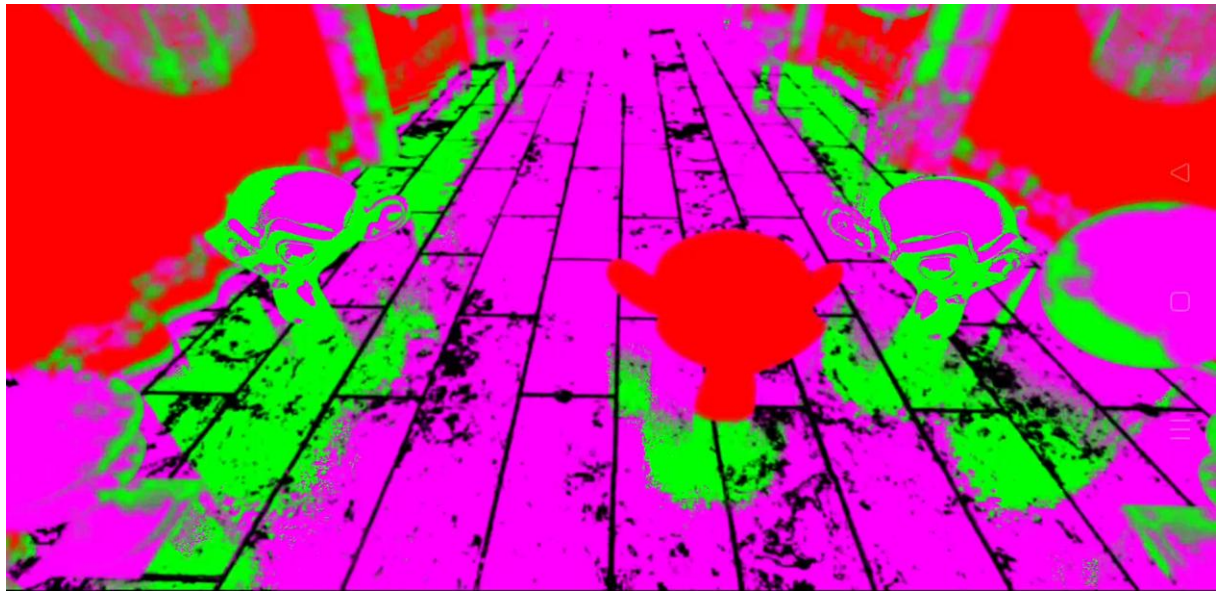
```
getGbufferData(svDispatchThreadId.xy, ...);
if(!isReflective())
{
  outputColor = reflectionFallback(...);
  return;
}
bool ssrHit = doSsrTrace(stochasticRayDir, ...);
if(ssrHit) {
  outputColor = sampleSsrColor(...);
}
else {
  uint writeOffset;
  InterlockedAdd(g_totalRqRays, 1, writeOffset);
  FailedSsr failedSSR = calculatedFailedPixel(...);
  g_failedSsr[writeOffset] = failedSsr;
  uint numRqRays = writeOffset + 1;
  uint groupCountX=(numRqRays+(RQ_THR_SIZE-1))/RQ_THR_SIZE;
  InterlockedMax(g_indirectArgs.m_groupCountX,groupCountX);
}
```

- PASS 2: Launch RQ
  - Use DispatchIndirect
  - Find reflection hit using RQ
  - Hit: Solve material secondary
    - RQ ray for direct illumination
  - Miss: reflect Sky

```
failedSsr = getFailedSsr(g_failedSsr[svDispatchThread.x]);
launchRqReflectionRays(failedSsr, ...);
if(rqHit) {
  launchRqShadowRay(rqHitData, ...);
  g_output[failedSsr.writeCoord] =
    calculateReflection(rqHitData, ...);
}
else {
  g_output[failedSsr.writeCoord] = reflectSky();
}
```

Hybrid (SSR+RQ) coverage





SSR Miss (RQ hit)

SSR Hit

Probe Fallback (No Ray)

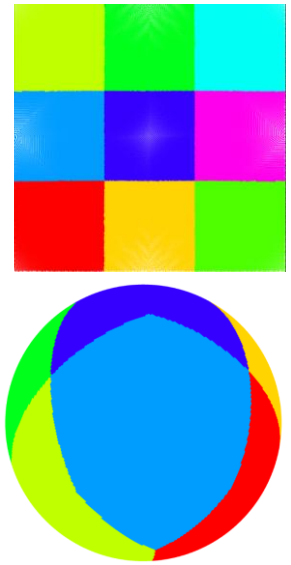
arm

Ray Binning

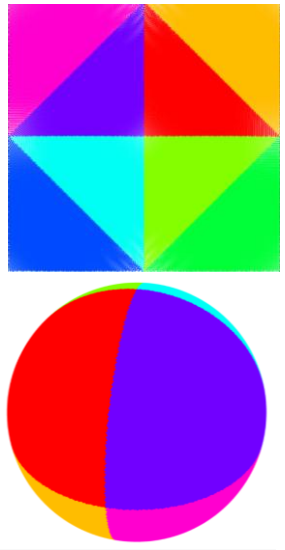
Reflections Optimizations

Ray binning

- SSR destroys coherence
  - Atomics execute in non defined order
  - Write unordered pixels
- We launch divergent RQ rays
  - Still affects performance
- Solution
  - We order pixels before writing
- Use ray direction:
  - usually more important than position
  - Multiple options
    - Octahedron bins, direction sign, etc.
- Objective: Order pixels that fail SSR using bins
  - Group by direction inside workgroup
  - Avoid a perfect order
  - Compact: Ensure no empty space



Octahedron Bins



Direction Sign

0-0 group 1 dir 1	0-1 group 1 dir 3	0-2 group 3 dir 1	0-3 group 3 dir 1
1-0 group 1 dir 1	1-1 group 1 dir NO	1-2 group 3 dir 2	1-3 group 3 dir 3
2-0 group 1 dir 2	2-1 group 1 dir 1	2-2 group 3 dir 3	2-3 group 3 dir 1
3-0 group 2 dir 3	3-1 group 2 dir 1	3-2 group 4 dir 1	3-3 group 4 dir NO
4-0 group 2 dir 1	4-1 group 2 dir 2	4-2 group 4 dir 1	4-3 group 4 dir 2
5-0 group 2 dir 2	5-1 group 2 dir 3	5-2 group 4 dir 2	5-3 group 4 dir 3

No ray Binning

Ray Binning

2-0 group 1 dir 2	0-2 group 3 dir 1
3-0 group 2 dir 3	2-3 group 3 dir 1
1-2 group 3 dir 2	0-3 group 3 dir 1
0-1 group 1 dir 3	1-3 group 3 dir 3
5-0 group 2 dir 3	2-2 group 3 dir 3
0-0 group 1 dir 1	1-2 group 3 dir 2
5-1 group 2 dir 3	4-2 group 4 dir 1
2-3 group 3 dir 1	3-2 group 4 dir 1
4-1 group 2 dir 2	4-3 group 4 dir 2
2-1 group 1 dir 1	5-2 group 4 dir 2
0-3 group 3 dir 1	5-3 group 4 dir 3
4-3 group 4 dir 2	0-0 group 1 dir 1
2-2 group 3 dir 3	2-1 group 1 dir 1
4-2 group 4 dir 1	1-0 group 1 dir 1
1-0 group 1 dir 1	2-0 group 1 dir 2
3-2 group 4 dir 1	0-1 group 1 dir 3
4-0 group 2 dir 1	4-0 group 2 dir 1
0-2 group 3 dir 1	3-1 group 2 dir 1
1-3 group 3 dir 3	5-1 group 2 dir 3
3-1 group 2 dir 1	3-0 group 2 dir 3
5-2 group 4 dir 2	4-1 group 2 dir 2
5-3 group 4 dir 3	5-0 group 2 dir 2

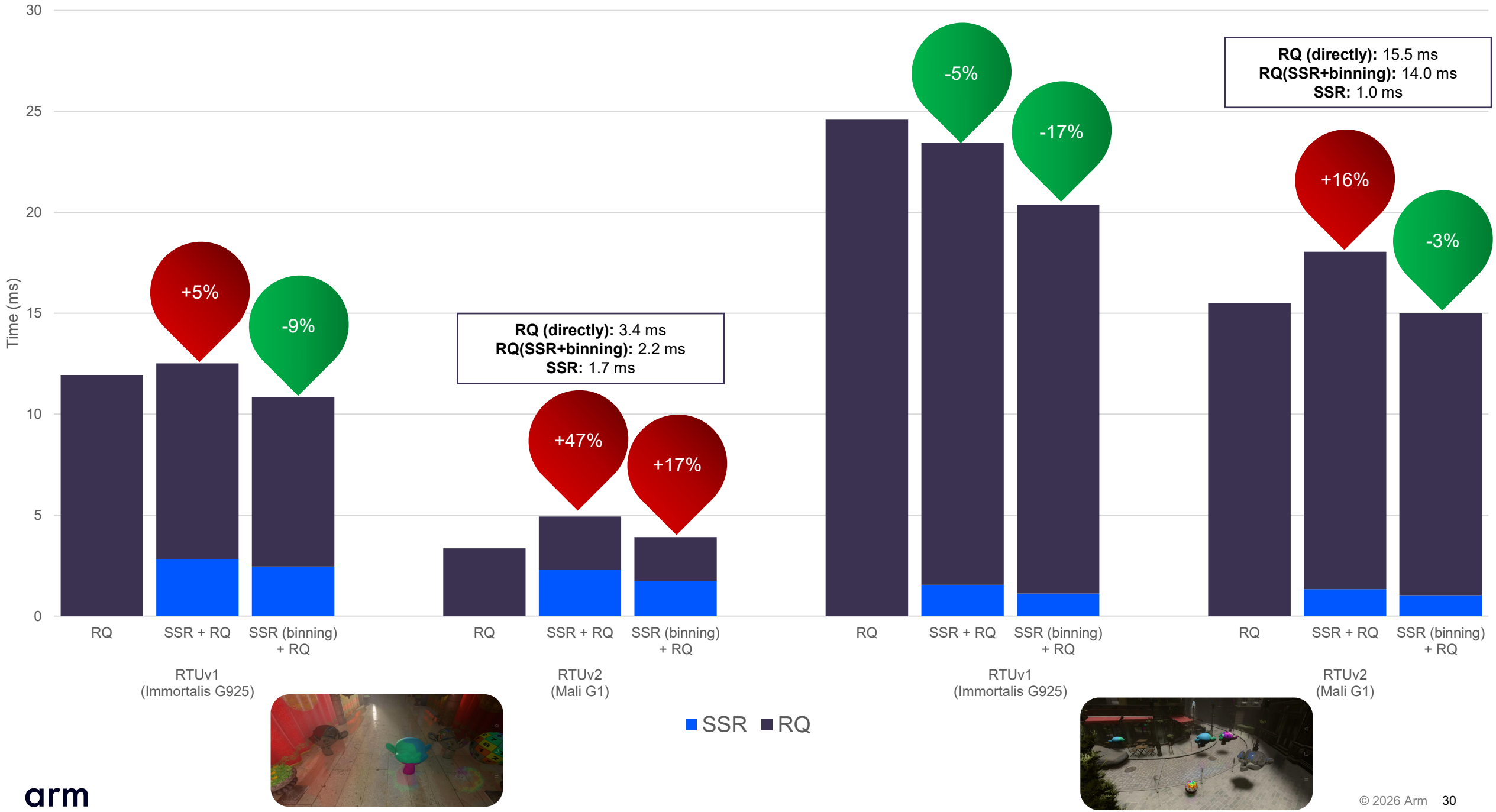
Ray binning

- For a SSR miss

1. Compute bin:
   - Use ray direction
2. Offset inside its group bin
   - Shared memory
   - Also, total number of rays for each group bin
   - Optimizations: pack, subgroup
3. Compute group global start
   - Atomic add to global counter
   - One thread per group
4. Final write position
   - Each thread
   - group start + offset inside group bin + size previous group bins

```
U32 calculateBin(Vec3 dirNorm) {
  #if BINNING_MODE_SIGN
  return ((dirNorm.z > 0) ? 4 : 0)
     + ((dirNorm.y > 0) ? 2 : 0) + ((dirNorm.x > 0) ? 1 : 0);
  #else if BINNING_MODE_OCTA
  Vec2 dirOctUv = octahedronEncode(dirNorm);
  IVec2 dirOctCoord = clamp(dirOctUv * Vec2(BIN_SIZE_X, BIN_SIZE_Y),
    IVec2(0,0), IVec2(BIN_SIZE_X-  1, BIN_SIZE_Y - 1));
  return dirOctCoord.x + dirOctCoord.y * BIN_SIZE_X;
  #else if BINNING_MODE_ETC
  //...
  #endif
}
```

```
U32 rayDirBin = calculateBin(reflDirNorm);
InterlockedAdd(gsBinCount[rayDirBin], 1, localBinIndex);
//WaveAtomicAdd(gsBinCount[rayDirBin/4] 1<<(8*(rayDirBin%4)),localBinIndex);
GroupMemoryBarrierWithGroupSync();
if(svGroupIndex == 0)
{
  U32 groupPreffixElements = 0;
  for(U32 b = 0; b < BIN_SIZE; ++b) {
    groupPreffixElements += gsBinCount[b];
  }
  InterlockedAdd(g_globalRqRays, groupPreffixElements, groupStart);
  gsGroupStart = groupStart;
  U32 threadGroupCountX = (groupStart
    + groupPreffixElements + (RQ_GROUP_SIZE - 1)) / RQ_GROUP_SIZE;
  InterlockedMax(g_rqIndirectArgs.m_threadGroupCountX, threadGroupCountX);
}
GroupMemoryBarrierWithGroupSync();
U32 writeOffset = gsGroupStart + localBinIndex;
for(U32 b = 0; b < BIN_SIZE; ++b) {
  writeOffset += gsBinCount[b];
}
g_pixelsFailedSsr[writeOffset] = calculatedFailedPixel(...);
```


30
25
20
15
10
5
0
Time (ms)
+5%
-9%
+47%
+17%
-5%
-17%
+16%
-3%
RQ (directly): 3.4 ms
RQ(SSR+binning): 2.2 ms
SSR: 1.7 ms
RQ (directly): 15.5 ms
RQ(SSR+binning): 14.0 ms
SSR: 1.0 ms
RQ
SSR + RQ
SSR (binning) + RQ
RTUv1 (Immortalis G925)
RTUv2 (Mali G1)
SSR
RQ

Hybrid (SSR+RQ) and ray binning conclusion

- SSR has a cost
  - Maybe greater than RQ savings
  - Might not be needed on simple scenes
  - More relevant for older hardware
    - RTU v1: Usually SSR helps
    - RTU v2: SSR might help on heavy scenes
- Ray binning
  - Might be key for proper SSR performance
  - Multiple options: content dependent → experiment
- Check your content:
  - Impact might vary
- Saving to SSR miss to buffer
  - Good warp occupancy
  - Huge ray divergence on RQ rays: atomic
    - We lose spatial ordering
    - Near pixels: similar world origin
- Ray coherence affects performance
  - Ray direction: usually more important than position

Less relevant on RTU v2

Better on heavy scenes

Combine with Ray binning

Content dependent

Buffer: Improves occupancy

Experiment and check your scene

arm

arm

Reflections Solving the Material

1. Ray tracing pipeline
2. RQ Uber Shader
3. RQ Visibility Buffer

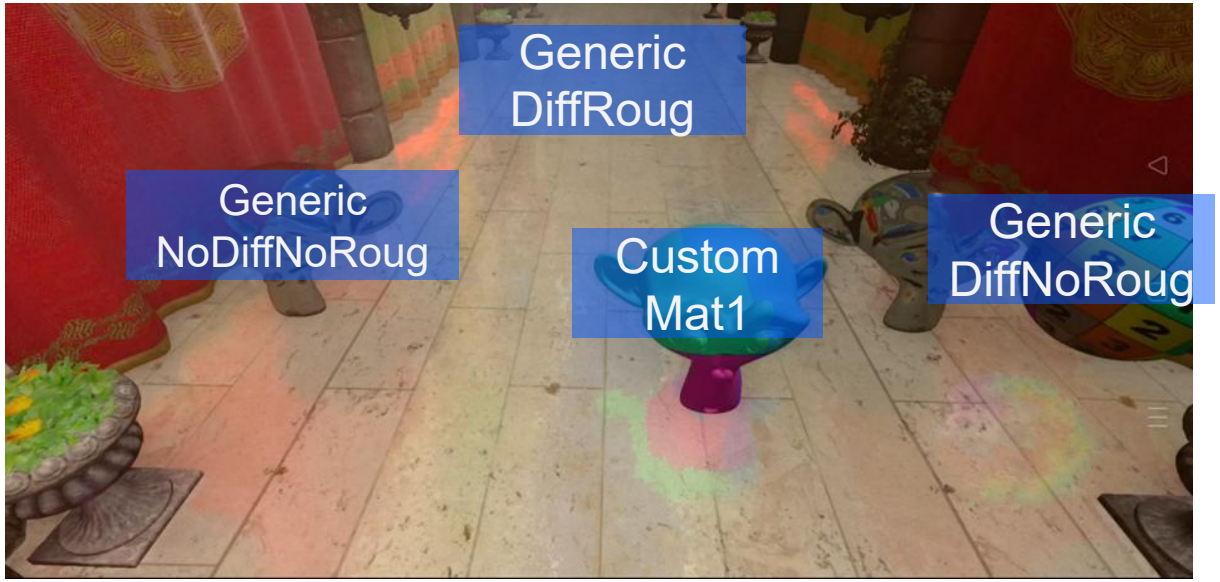
arm

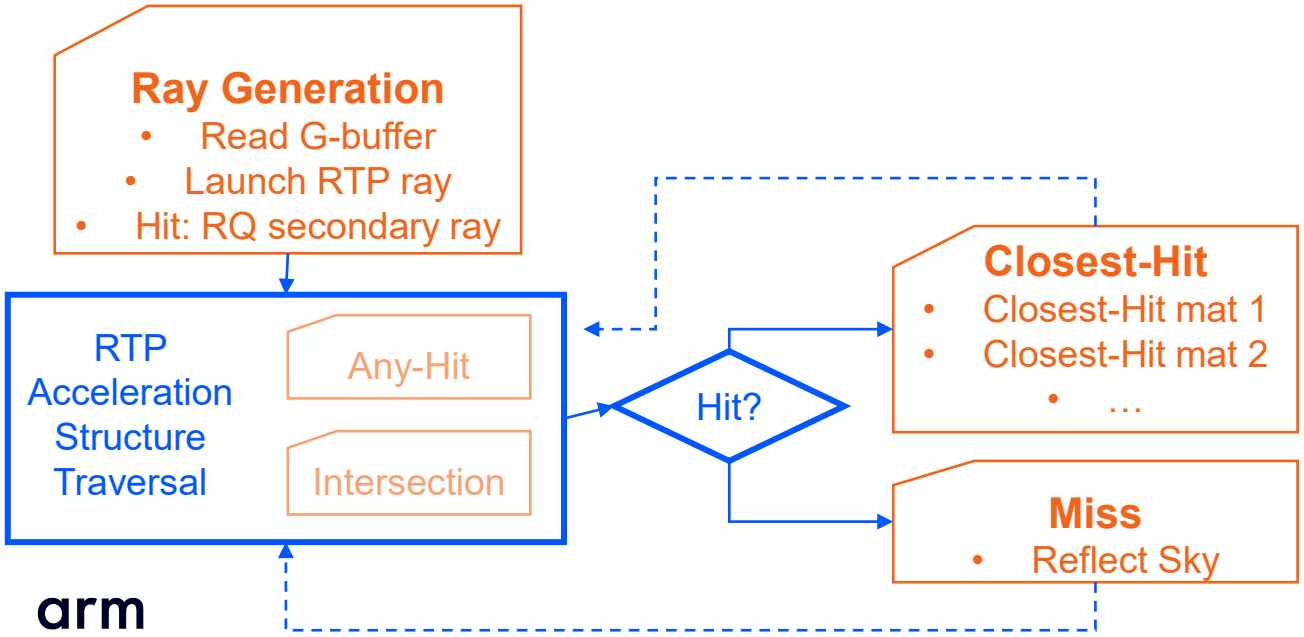
Ray Tracing Pipeline

Reflections Solving the Material

Ray tracing pipeline reflections

- Games usually have multiple materials
  - How to handle them in reflections
- RTP: Intended for many-material workloads.
  - Excessive shaders will disable inlining in Mali
  - One closest-hit per material
- Secondary rays: RQ in ray-gen:
  - Problematic on RTUv1 hardware
- Main problem: Still low support on mobile





arm

arm

Ray Query Uber Shader

Reflections Solving the Material

RQ Uber Shader

- How to solve materials in RQ
  - Uber Shader
- How our RQ uber shader works:
  1. Read failed-pixel
     - Dynamic dispatch → only SSR miss
     - write coords, direction, origin, etc.
  2. Launch RQ
  3. On hit:
     - Launch secondary RQ for direct light.
     - **Fetch material and illuminate**
       - Write coords from SSR miss data
  4. On miss: Sample sky
- Real scenes: multiple materials
  - Need to fetch correct material
  - Shader ID per shader type-permutation
    - Precomputed unique Id
    - GPU accesses it using bindless
      - TLAS `customIndex` to shader ID.
  - Uber shader
    - Switch-case using shader Id
    - Bindless to retrieve material data

```
failedSsr = getFailedSsr(g_failedSsr[svDispatchThreadId.x]);//SSR Miss
launchRqReflectionRays(failedSsr, ...);
if(rqHit) {
  launchRqShadowRay(rqHitData, ...);
  g_output[failedSsr.writeCoord] = calculateReflection(rqHitData, ...);
}
else {
  g_output[failedSsr.writeCoord] = reflectSky();
}
```

```
Vec3 shadeMat1(RqHitData hitData, ...) {
  Vec3 barycentric = hitData.barycentrics;//q.CommitedTriangleBarycentrics();
  uint primitiveIdx = hitData.triangleIndex;//q.CommittedPrimitiveIndex();
  MaterialData matData = g_bindlessMaterialData[instanceID];
  MeshData meshData = g_bindlessMeshData[instanceID];
  VertexData v0 = g_unifiedGeom[meshData.firstIdx + primitiveIdx * 3+ 0];
  VertexData v1 = g_unifiedGeom[meshData.firstIdx + primitiveIdx * 3+ 1];
  VertexData v2 = g_unifiedGeom[meshData.firstIdx + primitiveIdx * 3+ 2];
  return shadeDiffuseMat(hitData, v0, v1, v2, matData, ...);
}
```

```
Vec3 calculateReflectionColor(RqHitData hitData, ...) {
  uint instanceID = hitData.instanceId;//q.CommittedInstanceID()
  uint rqShaderId = g_RqShaderId[instanceID];
  switch (rqShaderId) {
    case 1:
    return shadeMat1(instanceID, ...);
    break;
    case 2:
    //...
  }
}
```

arm

General Optimization

- Shading cost can be significant
- Multiple materials → shading divergence
- Combine Shader permutations
  - Most materials have similar permutations:
  - `matData.albedoConst`
  - `readTexture(matData.albedoTexture, uv);`
  - Combine both switch entries
  - Separate permutation from shaderId
    - Bitmask: `MainRqShaderId | PermutationId`
  - Branch on permutation code
    - Most code remains common
    - `if(rqShaderId&HAS_DIF_TEX_MASK)`
- Combine common code
  - Most materials → Common code to fetch vertex data
    - `fetchVertexNormalAndUv`
  - Consider `KHR_ray_tracing_position_fetch`
    - Use case dependent
- Drop irrelevant permutations
  - Alpha mode: Hits are opaque
  - Unused combinations
- Profile frequently: Check your optimization

```
Vec3 shadeMat1(RqHitData hitData, ...) {
  fetchVertexNormalAndUv(hitData, uv, normal, ...);
  float3 albedo = matData.albedo;
  return shadeDiffuseMat1(hitData, uv, normal, ...);
}
Vec3 shadeMat2(RqHitData hitData, ...) {
  fetchVertexNormalAndUv(hitData, uv, normal, ...);
  float3 albedo = readTexture(matData.albedoTexture, uv);
  return shadeDiffuseMat2(hitData, uv, normal, ...);
}
Vec3 calculateReflectionColor(RqHitData hitData, ...) {
  //...
  switch (rqShaderId) {
    case 1:
    return shadeMat1(instanceID, ...);
    break;
    case 2:
    return shadeMat2(instanceID, ...);
    //...
```

```
Vec3 shadeMatGeneric(RqHitData hitData, uint rqShaderId, ...)
{
  Vec3 albedo;
  albedo = (rqShaderId & HAS_DIFF_TEX_MASK == 0)
    : matData.albedo ? readTexture(matData.albedoTexture, uv);
  }
  return shadeDiffuseMat1(hitData, uv, normal, ...);
}
Vec3 calculateReflectionColor(RqHitData hitData, ...) {
  //...
  fetchVertexNormalAndUv(hitData, uv, normal, ...);
  switch (rqShaderId & SHADER_ID_MASK) {
    case 1:
    return shadeMatGeneric(hitData, rqShaderId, ...);
    break;
    //...
```

arm

Problems with RQ uber shader

- RQ uber shaders are hard to maintain.
  - Generate Shader Ids
  - Group shaders in a switch
  - Additional build steps.
- Include all materials
  - Even those no currently used
- Shader size grows quickly
  - Increase entries in switch
  - SPIR-V Shader compilation increases.
  - Check frequently with **malioc**
    - Uber shader can have performance problems

```
Vec3 calculateReflectionColor(RqHitData hitData, ...) {
  uint instanceID = hitData.instanceId;//q.CommittedInstanceID()
  uint rqShaderId = g_RqShaderId[instanceID];
  switch (rqShaderId) {
    case MAT_001:
    return shadeMat001<0/*DIFF_TEX*/,0/*ROUGH_TEX*/,...>(...);
    break;
    case MAT_001_DIFF_ROUGH:
    return shadeMat001<1/*DIFF_TEX*/,1/*ROUGH_TEX*/,...>(...);
    case MAT_001_ROUGH:
    return shadeMat001<1/*DIFF_TEX*/,1/*ROUGH_TEX*/,...>(...);
    break;
    case MAT_001_DIFF_ROUGH:
    return shadeMat001<1/*DIFF_TEX*/,1/*ROUGH_TEX*/,...>(...);
    break;
    case MAT_002:
    return shadeMat002<0,1,...>(...);
    break;
    case MAT_003:
    return shadeMat003<0,0,...>(...);
    break;
    case MAT_137:
    return shadeMat137<0,0,...>(...);
    break;
    //...
  }
}
```

arm

Ray Query Visibility Buffer

Reflections Solving the Material

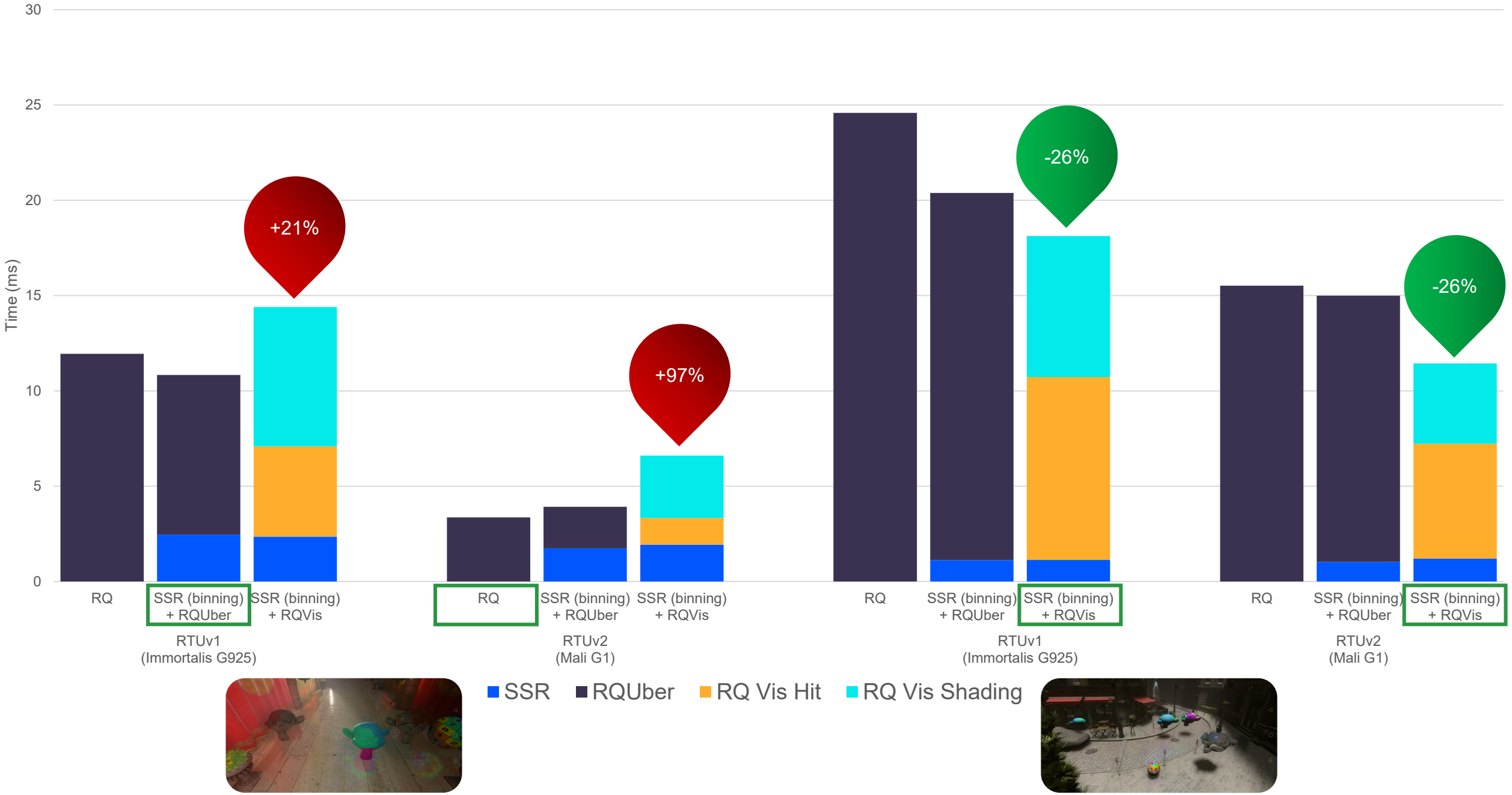
RQ visibility buffer

- RQ uber alternative
- Idea: Shade each material type in its own compute shader
  1. Launch RQ ray:
     - Do not shade the hit.
     - Store hit/miss data in intermediate buffer.
  2. Reorder intermediate buffer:
     - Sort by shader ID.
     - Produces coherent shading batches.
     - Compute batch sizes to dispatch dynamically.
     - See extra slides for details
       - Atomics can be optimized
  3. Shade in multiple passes:
     - One shader per material → Like GBuffer.
     - Dispatch indirect: Compute pass per shader id
     - Launch secondary RQ ray: Except miss
- Less shading divergence
  - RQ Miss Group: No Secondary RQ
  - Simpler pipeline management
    - Remove unused shaders

```
failedSsr =
  getFailedSsrPixel( g_pixelsFailedSsr[svDispatchThreadId.x]);
launchRqReflectionRay(failedSsr, ...);
g_hitPos[writeCoord] = worldPos + reflectionDir * hitData.rayT;
g_rqInstanceId[svDispatchThreadId.x] =
  rqHit ? hitData.instanceId ? 0xFF;
g_rqHitData[svDispatchThreadId.x].pdf_texLoD = ...;
g_rqHitData[svDispatchThreadId.x].realCoord = writeCoord;
g_rqHitData[svDispatchThreadId.x].rayDirX_or_barsX =
  rqhit ? hitData.bars.x : reflectionDir.x;
g_rqHitData[svDispatchThreadId.x].rayDirY_or_barsY =
  rqhit ? hitData.bars.y : reflectionDir.y;
g_rqHitData[svDispatchThreadId.x].rayDirZ_or_primitiveId =
  rqhit ? hitData.primitiveId : reflectionDir.z;
```


```
For each material
  if (svDispatchThreadId.x >= g_rqMatTotal[g_rqMatData.Id]) {
    return;
  }
  uint pixelIdxIdx = g_rqMatEnd[g_rqMatData.Id] - svDispatchThreadId.x;
  uint pixelIdx = g_orderedIndices[pixelIdxIdx];
  g_rqInstanceId[pixelIdx] = instanceId;
  hitData = g_rqHitData[pixelIdx];
  ShadeThisMat();
```

arm


30
25
20
15
10
5
0
Time (ms)
+21%
+97%
-26%
-26%
RQ
SSR (binning) + RQUber
SSR (binning) + RQVis
RQ
SSR (binning) + RQUber
SSR (binning) + RQVis
RQ
SSR (binning) + RQUber
SSR (binning) + RQVis
RQ
SSR (binning) + RQUber
SSR (binning) + RQVis
RTUv1 (Immortalis G925)
RTUv2 (Mali G1)
RTUv1 (Immortalis G925)
RTUv2 (Mali G1)
SSR
RQUber
RQ Vis Hit
RQ Vis Shading

arm

Conclusions

Conclusions

- Lots of room for optimizations in ray tracing content
  - Most are content dependent
  - Profile frequently
- Developers should reconsider previous ray tracing assumptions
  - Visual quality
  - Key differentiator
  - Ray tracing is a lot faster
    - Challenge conclusion from previous generations
    - Mali G1 offers a huge performance increase
  - More techniques are possible
  - We encourage developers to start experimenting
    - Profile and optimize
    - Check best practices
      - Mali: developer.arm.com/documentation/101897/latest

New techniques possible on mobile

Profile and optimize

Mali G1 significantly improves performance

Ray tracing offers significant visual improvements

arm

arm

Questions

arm

EXTRA: Ordering the RQ visibility Buffer

RQ visibility buffer – Ordering the buffer (count)

- We order the RQ hit/miss data per bucket
  - How to compute bucket:
    - `RqMiss ? 0 : (1+shaderId)`
- Three passes

1. **Count**: Get size of each bucket
   - Atomic increment on:
     - `g_bucketTotalElems[bucket]`
     - Start: 0
     - End: Total elements in the bucket
   - Dispatch: One thread per element
2. **Offset**: Calculate start of each bucket
3. **Reorder/copy**: Copy `threadId` in correct position

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
InterlockedAdd(g_bucketTotalElems[bucket], 1);
```

```
uint bucket = svDispatchThreadId.x;
if(bucket < g_consts.m_numBuckets)
{
  uint bucketSize = g_bucketTotalElems[bucket];
  g_indirectArgs[bucket].threadGroupCountX =
    (g_bucketTotalElems[bucket]+
    (THREADS_PER_GROUP-1)) / THREADS_PER_GROUP;
  g_indirectArgs[bucket].threadGroupCountY = 1;
  g_indirectArgs[bucket].threadGroupCountZ = 1;
  InterlockedAdd(g_globalAllBucketsSize,
    bucketSize, g_bucketOffset[bucket]);
}
```

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
U32 pixelOffset;
InterlockedAdd(g_bucketOffsets[bucket], 1, pixelOffset);
g_orderedIndices[pixelOffset] = svDispatchThreadId.x;
```

RQ visibility buffer – Ordering the buffer (offset)

1. **Count**: Get size of each bucket
2. **Offset**: Calculate start of each bucket
   - Atomically add all bucket sizes
   - Atomic on global `globalAllBucketsSize`
     - Start: 0
     - End: Total number of elements (unused)
     - Value before atomic:
       - `g_bucketOffset[bucket]`
       - Contains the start of the bucket
       - Number of elements before first bucket element
   - Set dispatch arguments
     - Shading will do one dispatch indirect per bucket
   - Dispatch: One thread per bucket
     - Almost immediate
3. **Reorder/copy**: Copy `threadId` in correct position

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
InterlockedAdd(g_bucketTotalElems[bucket], 1);
```

```
uint bucket = svDispatchThreadId.x;
if(bucket < g_consts.m_numBuckets)
{
  uint bucketSize = g_bucketTotalElems[bucket];
  g_indirectArgs[bucket].threadGroupCountX =
    (g_bucketTotalElems[bucket]+
    (THREADS_PER_GROUP-1)) / THREADS_PER_GROUP;
  g_indirectArgs[bucket].threadGroupCountY = 1;
  g_indirectArgs[bucket].threadGroupCountZ = 1;
  InterlockedAdd(g_globalAllBucketsSize,
    bucketSize, g_bucketOffset[bucket]);
}
```

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
U32 pixelOffset;
InterlockedAdd(g_bucketOffsets[bucket], 1, pixelOffset);
g_orderedIndices[pixelOffset] = svDispatchThreadId.x;
```

RQ visibility buffer – Ordering the buffer (copy)

1. **Count**: Get size of each bucket
2. **Offset**: Calculate start of each bucket
3. **Reorder/copy**: Copy `threadId` in correct position
   - Atomic increment:
     - `g_bucketOffsets[bucket]`
     - Start: Start of the bucket
     - End: End of the bucket
       - Number of element before last element in the bucket
       - Used in shading to get the element offset:
         - `g_bucketOffsets[g_bucket.Id] - svDispatchThreadId.x;`
     - Value before atomic:
       - `pixelOffset`
       - This contains the write position
         - Use it to store the `threadId`
       - Is the number of elements before current thread
   - Dispatch: One thread per element

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
InterlockedAdd(g_bucketTotalElems[bucket], 1);
```

```
uint bucket = svDispatchThreadId.x;
if(bucket < g_consts.m_numBuckets)
{
  uint bucketSize = g_bucketTotalElems[bucket];
  g_indirectArgs[bucket].threadGroupCountX =
    (g_bucketTotalElems[bucket]+
    (THREADS_PER_GROUP-1)) / THREADS_PER_GROUP;
  g_indirectArgs[bucket].threadGroupCountY = 1;
  g_indirectArgs[bucket].threadGroupCountZ = 1;
  InterlockedAdd(g_globalAllBucketsSize,
    bucketSize, g_bucketOffset[bucket]);
}
```

```
if(svDispatchThreadId.x >= g_elementsCount[0]) {
  return;
}
uint bucket = calculateRqShaderIdBucket(svDispatchThreadId.x);
U32 pixelOffset;
InterlockedAdd(g_bucketOffsets[bucket], 1, pixelOffset);
g_orderedIndices[pixelOffset] = svDispatchThreadId.x;
```

RQ visibility buffer – Ordering the buffer (optimizations)

- Requires atomics heavily.
  - Bad performance if not optimized
- Optimize atomics:
- **Basic version:**
  - Brute force:
  - Each thread increments its bins
  - Independent atomic buckets
    - In practice, most warps will execute the same atomics
    - Few bucket divergence on warp
    - Difficult to optimize by the driver
- **Loop-based version:**
  - Group atomics into same bucket
  - Use a loop with wave operations
  - All threads in the subgroup increment same atomic bucket at one
  - Make atomic uniformity more explicit
  - Good performance on other GPUs
  - Slower on Mali

```
# define UniformWaveAtomicCount2(dest, index, orig_value) \
{ \
  InterlockedAdd(dest[index], 1, orig_value); \
}
```

```
# define UniformWaveAtomicCount2(dest, index, orig_value) \
{ \
  bool _UniformWaveAtomicCount_bDone = false; \
  while(WaveActiveAnyTrue(!_UniformWaveAtomicCount_bDone)) \
  { \
    if(!_UniformWaveAtomicCount_bDone) \
    { \
      U32 _UniformWaveAtomicCount_waveIndex = \
         WaveReadLaneFirst(index); \
      if(index == _UniformWaveAtomicCount_waveIndex) \
      { \
        InterlockedAdd(dest[index], 1, orig_value); \
        _UniformWaveAtomicCount_bDone = true; \
      } \
    } \
  } \
}
```

arm

RQ visibility buffer – Ordering the buffer (optimizations)

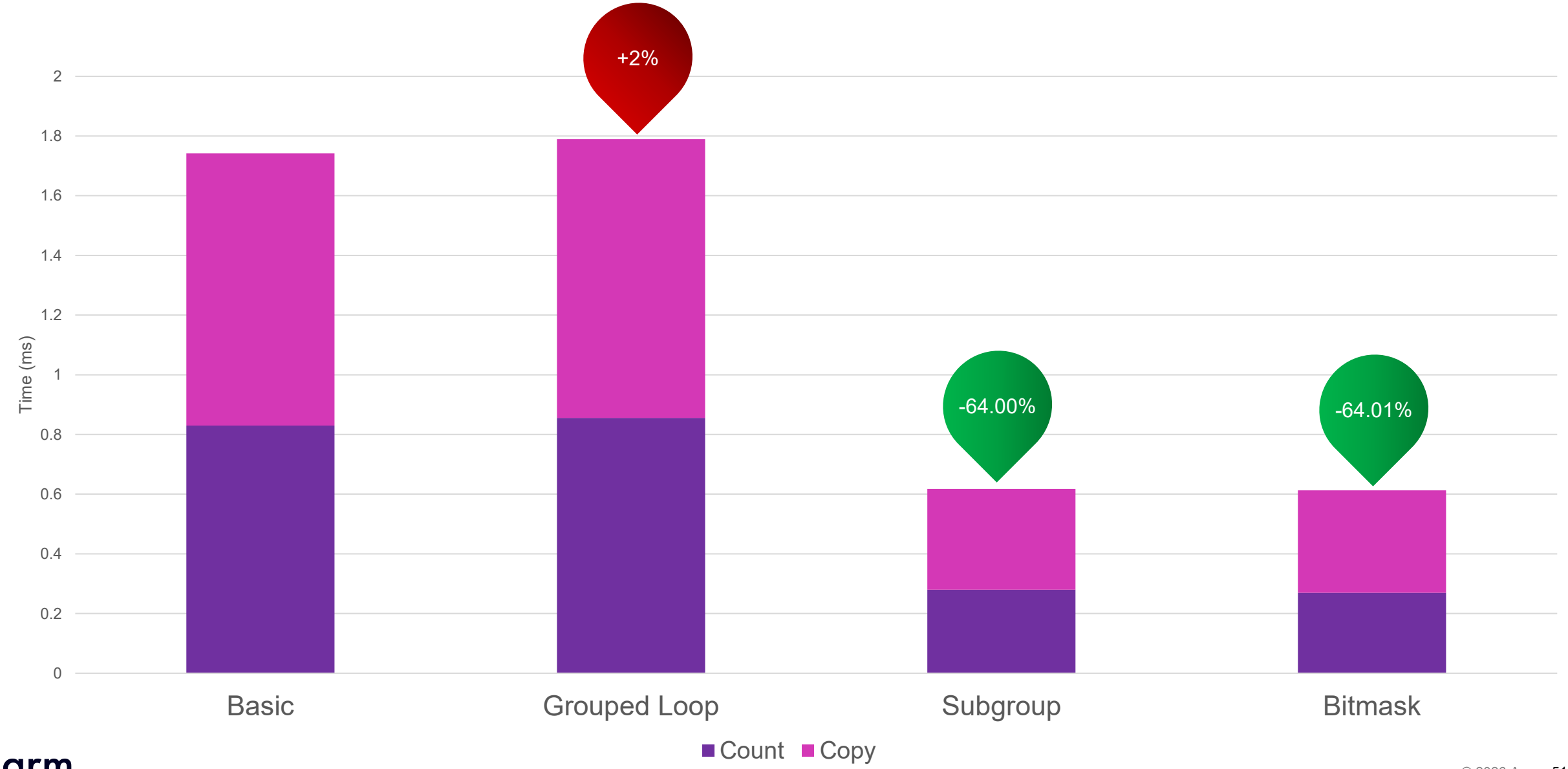
- **Subgroup version:**
  - Significant improvements in Mali.
  - Similar performance to loop based version on other GPUs
  - Use `WaveAtomics`
    - Subgroup operation to count all threads in the subgroup that target a bucket
    - Final atomic done by a single thread
    - We do one single atomic for each bucket in the subgroup
- **Bitmask wave** (no relevant)
  - Possible to optimize code further
  - Theoretically faster: Limited impact
  - Subgroup ballot optimization
    - `WaveActiveBallot`
  - We calculate all active threads at the beginning
    - Select lead thread using bitmask
    - Find threads with same bin
    - Update bin using wave operations
    - Update bit mask

```
# define WaveAtomicCount2(dest, orig_value) \
{ \
  uint _WaveAtomicCount_Value = WaveActiveCountBits(true); \
  if(WaveIsFirstLane()) \
  { \
    InterlockedAdd(dest, _WaveAtomicCount_Value, orig_value); \
  } \
  uint _WaveAtomCnt_PrefixValue=WavePrefixCountBits(true); \
  orig_value = WaveReadLaneFirst(orig_value); \
  orig_value = orig_value + _WaveAtomCount_PrefixValue; \
}

# define UniformWaveAtomicCount2(dest, index, orig_value) \
{ \
  bool _UniformWaveAtomicCount_bDone = false; \
  while(WaveActiveAnyTrue(!_UniformWaveAtomicCount_bDone)) \
  { \
    if(!_UniformWaveAtomicCount_bDone) \
    { \
      const U32 _UniformWaveAtomicCount_waveIndex = \
        WaveReadLaneFirst(index); \
      if(index == _UniformWaveAtomicCount_waveIndex) \
      { \
        /*InterlockedAdd(dest[index], 1, orig_value);*/ \
        WaveAtomicCount2(SBUFF(dest, index), orig_value); \
        _UniformWaveAtomicCount_bDone = true; \
      } \
    } \
  } \
}
```

arm

RQ visibility buffer – Ordering the buffer (performance)



arm

Tack
ಧನ್ಯವಾದಗಳು
Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank you
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm