TRAVE\textsc{R}ESER\textsc{CH}

Guest Lecture Utrecht University 2023
NEW PHONE WHO DIS?

00 Studied IGAD at the NHTV Breda University of Applied Sciences

01 Now a Rendering Engineer at Traverse Research. We research next-gen graphics with a focus on Ray Tracing.

02 I work mostly on algorithms on the GPU and optimizing them. Working closely together with AMD.

03 Most CPU programming is in Rust, and all GPU programming is in HLSL.

04 In my free time I write software rasterizers thanks to Jacco
**TODAY’S SCHEDULE**

00  AMD GPU Overview

01  Analysing GPU performance with the AMD Radeon Developer Suite

02  Some Practical Examples

03  Some time for Q&A

WHAT ARE WE GOING TO DO TONIGHT BRAIN?

THE SAME THING WE DO EVERY NIGHT, PINKY, TRY TO TAKE OVER THE WORLD.
So far Jacco has only shown Nvidia GPUs, so let’s take a look at an AMD GPU today.

**AMD RX 7900 XTX with RNDA3 Architecture and Chiplet Design**

- 24 GiB of GDDR6 Memory at 960 GiB/s
- 61.42 TFLOPS for FP32

Warps will be called ‘waves’ today. And we’ll call threads ‘lanes’.

Sources:
- https://www.techpowerup.com/gpu-specs/radeon-rx-7900-xtx.c3941
- https://chipsandcheese.com/2023/01/07/microbenchmarking-amds-rdna-3-graphics-architecture/
LET’S ZOOM IN A LITTLE BIT

00 6 Shader Engines

01 6 Chiplets with each 16 MiB of Infinity Cache, for a total of 96 MiB. Bandwidth: ~3.3 TiB/s, Latency: ~166 ns

02 6 Mib of Shared L2 Cache. Bandwidth: ~8.7 TiB/s, latency: ~87 ns

03 Various other blocks for rasterization, video encoding/decoding, PCI-E, Outputting Display

WARNING!
These diagrams are only an indication of what the hardware roughly looks like. IHVs don’t like giving away details, so this is all we get. Just take everything with a grain of salt.
Each shader engine consists of 8 Work Group Processors (WGP), laid out in 2 Shader Arrays.

Each Shader Array gets a shared L1 cache of 256 KiB. Bandwidth: ~17 TiB/s, latency: ~49 ns.

And more blocks for rasterication (RB+, Rasterizer, Primitive Unit).
WE NEED TO GO DEEPER
Each WGP has 4 SIMDs, laid out in 2 Compute Units.

Each Compute Unit gets a 32 KiB block of L0 cache. 
Bandwidth: ~30 TiB/s, latency: ~33 ns

32 KiB Instruction Cache
16 KiB Scalar Cache
128 KiB Local Data Shared (groupshared, __local)

Texture Filtering and Texture Mapping Units

Where's the Ray Tracing?
2048 Scalar General Purpose Registers (SGPRs)
1536 Vector General Purpose Registers (VGPRs)

Only 1024 VGPRs on last gen and smaller GPUs. May not get 1536 VGPRs in future hardware.

Scheduler can hold 16 waves in flight. Each wave can hold a variable amount of VGPRs, SGPRs, and LDS.

If we want to have 16 waves in flight, each wave gets:
- $\frac{1536}{16} = 96$ VGPRs
- $\frac{2048}{16} = 128$ SGPRs

Spoiler alert: Only half of that in other cases!

32 lanes * 16 waves * 4 SIMDs * 8 WGP * 6 SEs = 98304 lanes in flight! 😲
WHAT IS SCALAR AND WHAT IS VECTOR?

01 Similar to programming with SIMD on a CPU where we also have scalar and vector.

02 Scalar Registers hold values that are the same for each lane in a wave.

03 Vector Registers hold values that are different for each lane in the wave.

04 This gives the programmer more registers to work with, without needing a lot more silicon.

05 You can roughly think of a VGPR as an __m1024

```
int     -> scalar
float   -> scalar
__m128i -> vector
__m256  -> vector
```

```
flo44  value = buffer0[threadID];
flo44  multiplier = buffer1[groupID];
flo44  adder = buffer2[threadID];

for (int i = 0; i < constants.loopCount; ++i) {
    value = value * multiplier + adder;
}
```

// blue is scalar
// red is vector
A shader/kernel can run in either wave32 or wave64 mode.

In wave64 mode, 2 VGPRs registers are used for each value, and each instruction is issued twice.

If we want to have 16 waves in flight, each wave now gets:
- \((1536 / 2) / 16 = 48\) VGPRs
- \(2048 / 16 = 128\) SGPRs

In some cases wave32 is more efficient, in others wave64 is better. The compiler decides for you.

```c
__m1024[2] a, b, c;
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
```

64 lanes * 16 waves * 4 SIMDs * 8 WGs * 6 SEs = 196608 lanes in flight! 🤯🤯🤯
AMD Radeon Developer Suite

AMD has some cool tools to help analyze how your code runs on an AMD GPU.

You can download them for free: https://gpuopen.com/introducing-radeon-developer-tool-suite/

Today we will be looking at the Radeon Graphics Profiler (RGP) and the Radeon GPU Analyzer (RGA)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor congue massa.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor congue massa.
This shader seems to be running at low occupancy. Let's investigate!

Looks like we are limited by VGPRs. Let's take a look at the instruction trace.

---

**Dispatch properties**

- Total thread groups: 520, 180, 1
- Thread group dimensions: 8, 8, 1
- Strict shader processor interpolator (SPI) ordering: OFF

**Wavefronts and threads**

- Total wavefronts: 57,600
- Total threads: 3,886,400
- Average wavefront duration: 4.349 µs
- Average threads per wavefront: 64

**Per-wavefront resources**

- Vector registers: 136 (144 allocated)
- Scalar registers: 40 (128 allocated)
- Registers spilled to scratch memory: OFF
- Local data share per thread group: OFF

**Theoretical wavefront occupancy**

The occupancy of this shader is limited by its vector register usage. This shader could potentially run 9 wavefronts out of 16 wavefronts per SIMD.

However, if you reduce vector register usage by 16 you could run another wavefront.
A couple thousand more VALU ops follow...

RGP says we are bound by the VALU unit.
AMD documents their ISA, so you can look up what all the instructions do. (Even the ones that are not exposed in HLSL, OpenCL, SPIR-V, etc.)

PRACTICAL EXAMPLES

00 Roughly calculating how much data can we process in a given time.

01 Finding hidden store instructions in your code.

02 Optimizing the LDS memory usage.

03 Crossing the streams. Letting the lanes communicate.
2 + 2 is 4 minus 1 is 3 QUICK MATHS

00 Remember the VRAM bandwidth?
960 GiB/s = 960 MiB/ms

01 With 16 ms in a frame, we can process ~15.4 GiB per frame at 60 Hz.

02 With some basic maths we can figure out how fast something could run before we hit a physical limitation.
**PRACTICAL EXAMPLE: PARTICLE SYSTEM**

00. Each particle has a position and velocity: 24 bytes

01. Let's say we want to update $64 \times 1024 \times 1024$ particles. That's 1536 MiB of data.

02. We can process 960 MiB of data in one ms.

03. $\frac{1536 \text{ MiB}}{960 \text{ MiB/ms}} = 1.6 \text{ ms}$

04. If you are profiling your code and you find that you are way slower than that number, you have another bottleneck somewhere.

05. We can do the same for ALU:

   $6 \text{ FP32 ops} \times 64 \times 1024 \times 1024 \text{ particles} = 400 \text{ million ops}$

   $\frac{400 \text{ MOP}}{61.42 \text{ GFLOP/ms}} = 6.2 \mu s$

   We will not be ALU bound in this case.
PRACTICAL EXAMPLE: HIDDEN STORES

This shader writes 36 DWORDs per lane to global memory.

But we wrote only one store in our code!

Let's go find the others!

_kernel void src_MyKernel(__global uint* input, __global uint* output) {
    uint threadID = get_global_id(0);
    uint array[32];
    uint a = 0, b = 1;
    for (uint i = 0; i < 32; ++i) {
        uint c = a + b;
        uint x = input[(threadID + c) & 0xffff];
        c += x & 0xf;
        array[i] = c;
        a = b;
        b = c;
    }
    output[threadID] = array[threadID % 32];
}
**PRACTICAL EXAMPLE: HIDDEN STORES**

This shader writes 36 DWORDs per lane to global memory.

But we wrote only one store in our code!

Let's go find the others!

```c
__kernel void src_MyKernel(__global uint* input, __global uint* output)
{
    uint threadID = get_global_id(0);
    uint array[32];
    uint a = 0, b = 1;
    for (uint i = 0; i < 32; ++i) {
        uint c = a + b;
        uint x = input[(threadID + c) & 0xffff];
        c += x & 0xff;
        array[i] = c;
        a = b;
        b = c;
    }
    output[threadID] = array[threadID % 32];
}
```

```assembly
0x01698 scratch_store_b128 off, v[0:3], s33 offset:4
0x016A0 scratch_store_b128 off, v[4:7], s33 offset:20
0x016A8 scratch_store_b128 off, v[8:11], s33 offset:36
0x016B0 scratch_store_b128 off, v[12:15], s33 offset:52
0x016B8 v_shlrev_b32_e32 v0, 2, v35
0x016BC s_clause 0x3
0x016C0 scratch_store_b128 off, v[16:19], s33 offset:68
0x016C8 scratch_store_b128 off, v[20:23], s33 offset:84
0x016D0 scratch_store_b128 off, v[24:27], s33 offset:100
0x016D8 scratch_store_b128 off, v[28:31], s33 offset:116
0x016E0 scratch_load_b32 v2, v0, s33 offset:4
0x016E8 v_lshr_uw_b64 v[0:1], 2, v[32:33] | instid0(VALU_DEP_1) | instskip(NEXT) | instid1(VALU_DEP_2)
0x016F4 v_add_co_u32 v0, vcc_lo, s2, v0
0x016FC v_add_co_ci_u32_e32 v1, vcc_lo, s3, v1, vcc_lo
0x01700 s_waitcnt vmcnt(0)
0x01704 global_store_b32 v[0:1], v2, off
0x0170C s_sendmsg sendmsg(MSG_DEALLOC_VGPRS)
0x01710 s_endpm
```
PRACTICAL EXAMPLE: AABB PAIR FINDING

00 Loop that iterates over neighbouring AABBs and finds the AABB with the smallest 'distance'.

01 Uses very clever bit packing and atomic operations to the minimal number of tests.

02 What I would like to focus on right now is the 'loadLdsAabb' function.

03 By changing this function we can change runtime from 2.77 ms to 2.43 ms.

04 But how?! It just loads some data from LDS!

```cpp
// Clear neighbors
gs_neighbors[localId] = ~0u;

groupMemoryBarrierWithGroupSync();

// Search for nearest neighbors (based on AABB SAH)
if (localId >= ldsMinBound && localId < ldsMaxBound) {
    uint min_area_index = ~0u;
    for (int r = 1; r <= kSearchRadius && localId + r < ldsMaxBound; ++r) {
        const float area = distance(aabb, loadLdsAabb(localId + r));
        const uint area_i = (asuint(area) << 1) & ~kEncodeMask;
        const uint encode0 = area_i | encodeRelativeOffset(r);
        const uint encode1 = area_i | encodeRelativeOffset(-r);
        min_area_index = min(min_area_index, encode0);
        InterlockedMin(gs_neighbors[localId + r], encode1);
    }
    InterlockedMin(gs_neighbors[localId], min_area_index);
}
groupMemoryBarrierWithGroupSync();
```
JUST ADD A LITTLE BIT OF TRANSPOSITION MAGIC

```c
// Array of Structures (AoS)

groupshared Aabb gs_aabbs[kAabbCount];
groupshared uint gs_neighbors[kAabbCount];
Aabb LoadLdsAabb(uint index) {
    Aabb aabb;
    aabb.minBounds.x = gs_aabbs[index * 6 + 0];
    aabb.minBounds.y = gs_aabbs[index * 6 + 1];
    aabb.minBounds.z = gs_aabbs[index * 6 + 2];
    aabb.maxBounds.x = gs_aabbs[index * 6 + 3];
    aabb.maxBounds.y = gs_aabbs[index * 6 + 4];
    aabb.maxBounds.z = gs_aabbs[index * 6 + 5];
    return aabb;
}
```

```c
// Structure of Arrays (SoA)

groupshared float gs_aabbs[6 * kAabbCount];
groupshared uint gs_neighbors[kAabbCount];
Aabb LoadLdsAabb(uint index) {
    Aabb aabb;
    aabb.minBounds.x = gs_aabbs[0 * kAabbCount + index];
    aabb.minBounds.y = gs_aabbs[1 * kAabbCount + index];
    aabb.minBounds.z = gs_aabbs[2 * kAabbCount + index];
    aabb.maxBounds.x = gs_aabbs[3 * kAabbCount + index];
    aabb.maxBounds.y = gs_aabbs[4 * kAabbCount + index];
    aabb.maxBounds.z = gs_aabbs[5 * kAabbCount + index];
    return aabb;
}
```
### LDS Bank Conflicts

#### Example from AMD

<table>
<thead>
<tr>
<th>Float4 array</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w</td>
</tr>
</tbody>
</table>

8 Conflicts vs 2 Conflicts

#### Array of floats

<table>
<thead>
<tr>
<th>Array of floats</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x</td>
</tr>
</tbody>
</table>

AOS vs SAO also matters on the GPU!

These days Waveops are available in DX12, Vulkan, and CUDA.

Lanes in a group could already work together via LDS, but bandwidth within one lane is much higher.

8.6 TB/s on LDS vs 51.6 TB/s on registers. This is on older GCN hardware. Couldn't find more recent stats 😞

Really opens up the possibility for new algorithms on the GPU.

```c
const uint nodesToAllocate = WaveActiveCountBits(1);
const uint prefixSum = WavePrefixCountBits(1);

uint baseAllocationIndex = 0;
if (WaveIsFirstLane()) {
    baseAllocationIndex = atomicCounters.interlockedAddUniform(kAtomicCounterInternalNodes, nodesToAllocate);
} baseAllocationIndex = WaveReadLaneFirst(baseAllocationIndex);
const uint allocationIndex = baseAllocationIndex + prefixSum;
```
These days Waveops are available in DX12, Vulkan, and CUDA.

Lanes in a group could already work together via LDS, but bandwidth within one lane is much higher.

8.6 TB/s on LDS vs 51.6 TB/s on registers. This is on older GCN hardware. Couldn't find more recent stats 😞

Really opens up the possibility for new algorithms on the GPU.

WARNING
Don't actually do this. The compiler generated faster code without it.

source: https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/
Wave Query
The intrinsics for querying a single wave.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Pixel shader</th>
<th>Compute shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaveGetLaneCount</td>
<td>Returns the number of lanes in the current wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveGetLaneIndex</td>
<td>Returns the index of the current lane within the current wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveIsFirstLane</td>
<td>Returns true only for the active lane in the current wave with the smallest index.</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Wave Vote
This set of intrinsics compare values across threads currently active from the current wave.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Pixel shader</th>
<th>Compute shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaveActiveAnyTrue</td>
<td>Returns true if the expression is true in any active lane in the current wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveAllTrue</td>
<td>Returns true if the expression is true in all active lanes in the current wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveBallot</td>
<td>Returns a 64-bit unsigned integer bitmask of the evaluation of the Boolean expression for all active lanes in the specified wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Wave Broadcast
These intrinsics enable all active lanes in the current wave to receive the value from the specified lane, effectively broadcasting it. The return value from an invalid lane is undefined.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Pixel shader</th>
<th>Compute shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaveReadLaneAll</td>
<td>Returns the value of the expression for the given lane index within the specified wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveReadLaneFirst</td>
<td>Returns the value of the expression for the active lane of the current wave with the smallest index.</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Wave Reduction
These intrinsics compute the specified operation across all active lanes in the wave and broadcast the final result to all active lanes. Therefore, the final output is guaranteed uniform across the wave.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Pixel shader</th>
<th>Compute shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaveActiveAllEqual</td>
<td>Returns true if the expression is the same for every active lane in the current wave (and thus uniform across it).</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveBitAnd</td>
<td>Returns the bitwise AND of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveBitOr</td>
<td>Returns the bitwise OR of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveBitXor</td>
<td>Returns the bitwise Exclusive OR of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveCountBits</td>
<td>Counts the number of boolean variables which evaluate to true across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveMax</td>
<td>Computes the maximum value of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveMin</td>
<td>Computes the minimum value of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveProduct</td>
<td>Multiplies the values of the expression together across all active lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WaveActiveSum</td>
<td>Sums up the value of the expression across all active lanes in the current wave and replicates it to all lanes in the current wave, and replicates the result to all lanes in the wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Wave Scan and Prefix
These intrinsics apply the operation to each lane and leave each partial result of the computation in the corresponding lane.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Pixel shader</th>
<th>Compute shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>WavePrefixCountBits</td>
<td>Returns the sum of all the specified boolean variables set to true across all active lanes with indices smaller than the current lane.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WavePrefixSum</td>
<td>Returns the sum of all of the values in the active lanes with smaller indices than this one.</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WavePrefixProduct</td>
<td>Returns the product of all of the values in the lanes before this one of the specified wave.</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

12.5.2. LDS Lane-permute Ops

DS_PERMUTE instructions allow data to be swizzled arbitrarily across lanes. Two versions of the instruction are provided: forward (scatter) and backward (gather). These exist in LDS only, not GDS.

Note that in wave64 mode the permute operates only across 32 lanes at a time on each half of a wave64. In other words, it executes as if were two independent wave32's. Each half-wave can use indices in the range 0-31 to reference lanes in that same half-wave.

These instructions use the LDS hardware but do not use any memory storage, and may be used by waves that have not allocated any LDS space. The instructions supply a data value from VGPRs and an index value per lane.

12.5. Data Share Indexed and Atomic Access

"RDNA3" Instruction Set Architecture

- \( \text{ds\_permute\_b32 : Dst[index(0..31)] = src[0..31]} \) Where [0..31] is the lane number
- \( \text{ds\_bpermute\_b32 : Dst[0..31] = src[index[0..31]]} \)

The EXE mask is honored for both reading the source and writing the destination. Index values out of range wrap around (only index bits [6:2] are used, the other bits of the index are ignored). Reading from disabled lanes returns zero.

In the instruction word: VDST is the dest VGPR, ADDR is the index VGPR, and DATAO is the source data VGPR. Note that index values are in bytes (so multiply by 4), and have the 'offset' field added to them before use.
**PRACTICAL EXAMPLE: PREFIX SUM COMPACTION**

00. The Prefix Sum is a fairly common operation that is not trivial to scale to many threads efficiently.

01. Waveops allow us to speed-up this process by first computing prefix sums per wave.

02. By also computing the sum we can then compute the per-group prefix sum.

03. Most operations now happen within the wave, which saves us from going to LDS Memory.

```c
const uint nodeIndex = (localId < clustersInChunk) ? bnd.clusterIndexMapB.loadUniform<uint>(rangeBase + baseClusterIndex + localId) : kInvalidCluster;
const bool clusterActive = nodeIndex != kInvalidId;
const uint partialPrefixSum = WavePrefixCountBits(clusterActive);
const uint partialSum = WaveActiveCountBits(clusterActive);

if (WaveIsFirstLane()) {
    gs_ldsBlock[waveId] = partialSum;
}

GroupMemoryBarrierWithGroupSync();

// Compute the prefix sum of the wave totals
ASSERT(kWavesInGroup <= kWaveSize); // would need more iterations if false
if (localId < kWavesInGroup) {
    const uint prefixSum = WavePrefixSum(gs_ldsBlock[localId]);
    const uint sum = WaveActiveSum(gs_ldsBlock[localId]);

    // implicit sync since this runs in one wave
    gs_ldsBlock[localId] = prefixSum;
}

GroupMemoryBarrierWithGroupSync();

if (localId < clustersInChunk && nodeIndex != kInvalidCluster) {
    const uint localPrefixSumBase = gs_ldsBlock[waveId];
    const uint newIndex = rangeBase + sumClustersInPrevChunks + localPrefixSumBase + partialPrefixSum;
    bnd.clusterIndexMapA.storeUniform<uint>(newIndex, nodeIndex);
}
THANK YOU

Also: We're hiring!
JOIN THE YELLOW SIDE

00 We are hiring for thesis students!

01 Located in the beautiful city of Breda, within walking distance of the train station.

02 Join a team of wonderful people and learn from each other.

03 We don't only do this close to hardware optimization of code, but also developing new algorithms with very complex math.

04 Real-Time Ray Tracing, Monte Carlo Rendering, ReSTIR, Denoising, Volumetrics, Machine Learning, etc