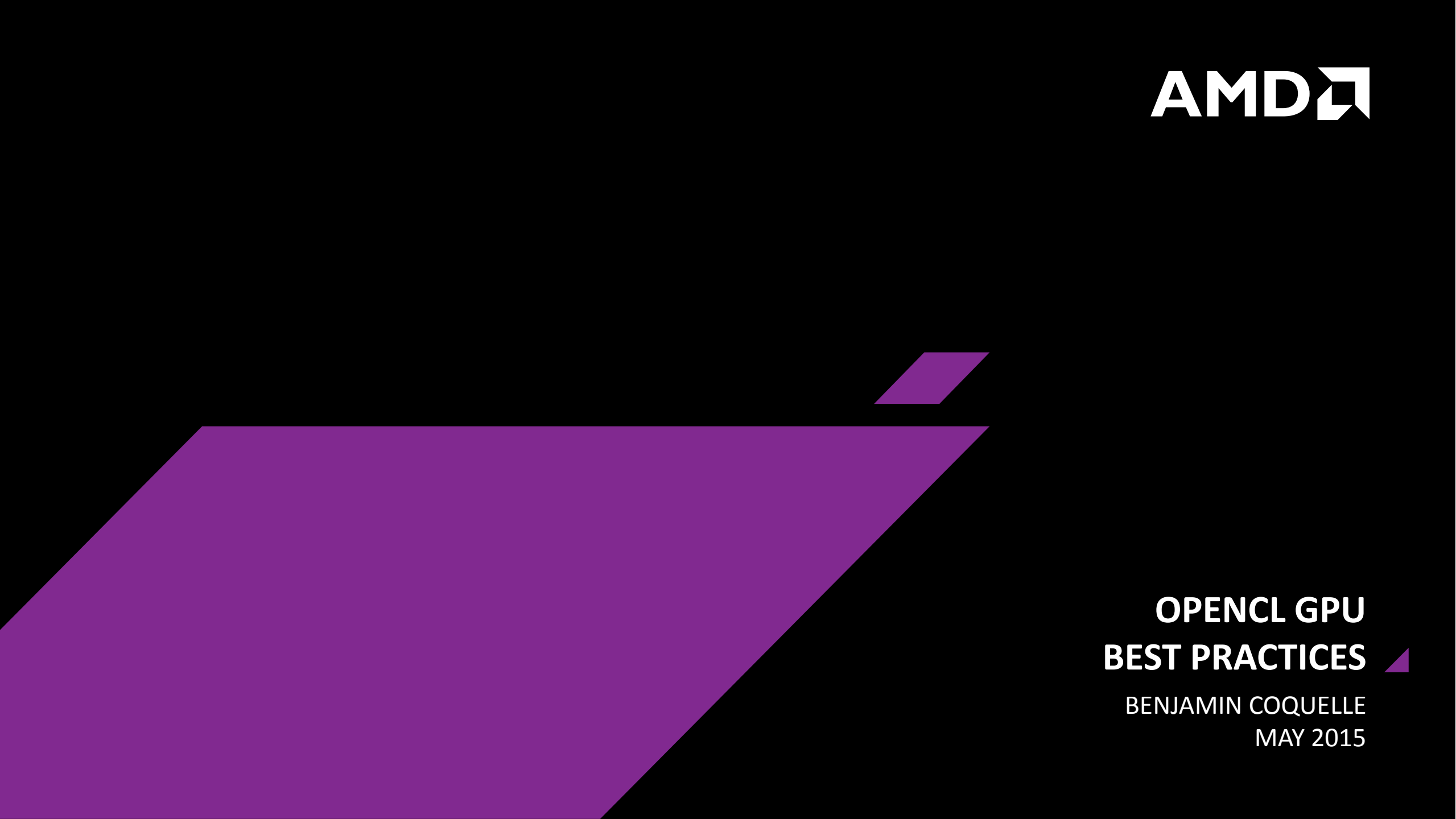


OPENCL GPU BEST PRACTICES

BENJAMIN COQUELLE

MAY 2015

TOPICS

- Data transfer
- Parallelism
- Coalesced memory access
- Best work group size
- Occupancy
- branching

- All the performance numbers come from a W8100 running on a 14.502.1019 driver

DATA TRANSFER

AMD

- To transfer data to the GPU, data need to be page locked. This operation is called pinning and it is costly (CPU time).
  - Therefore, by default, each time you call a data transfer function, we need to pin the host buffer :
    - clEnqueueWriteBuffer(queue, devicebuffer, …, **hostbuffer**, …)
    - clEnqueueReadBuffer(queue, devicebuffer, …, **hostbuffer**, …)
- OpenCL provides a mechanism to “pre-pinned” a buffer and thus achieve the best transfer rate on the PCIE bus

1. pinnedBuffer = clCreateBuffer(CL_MEM_ALLOC_HOST_PTR or CL_MEM_USE_HOST_PTR)
2. deviceBuffer = clCreateBuffer()
3. void *pinnedMemory = clEnqueueMapBuffer(pinnedBuffer) //pinning cost is incurred here
4. clEnqueueRead/WriteBuffer(deviceBuffer, pinnedMemory)
5. clEnqueueUnmapMemObject(pinnedBuffer, pinnedMemory)

- Typically an application will perform step 1, 2 ,3 and 5 once. While the mapped pinned buffer can be uploaded several times from the CPU and thus different data can be uploaded while repeating step 4

DATA TRANSFER

PERFORMANCE RESULT

AMD

Write operation

method	0.5MB	1MB	10MB	100MB
Pre-pinned	5GB/s	7.5GB/s	12GB/s	12.5GB/s
classic	2.1GB/s	3GB/s	6.3GB/s	6.8GB/s

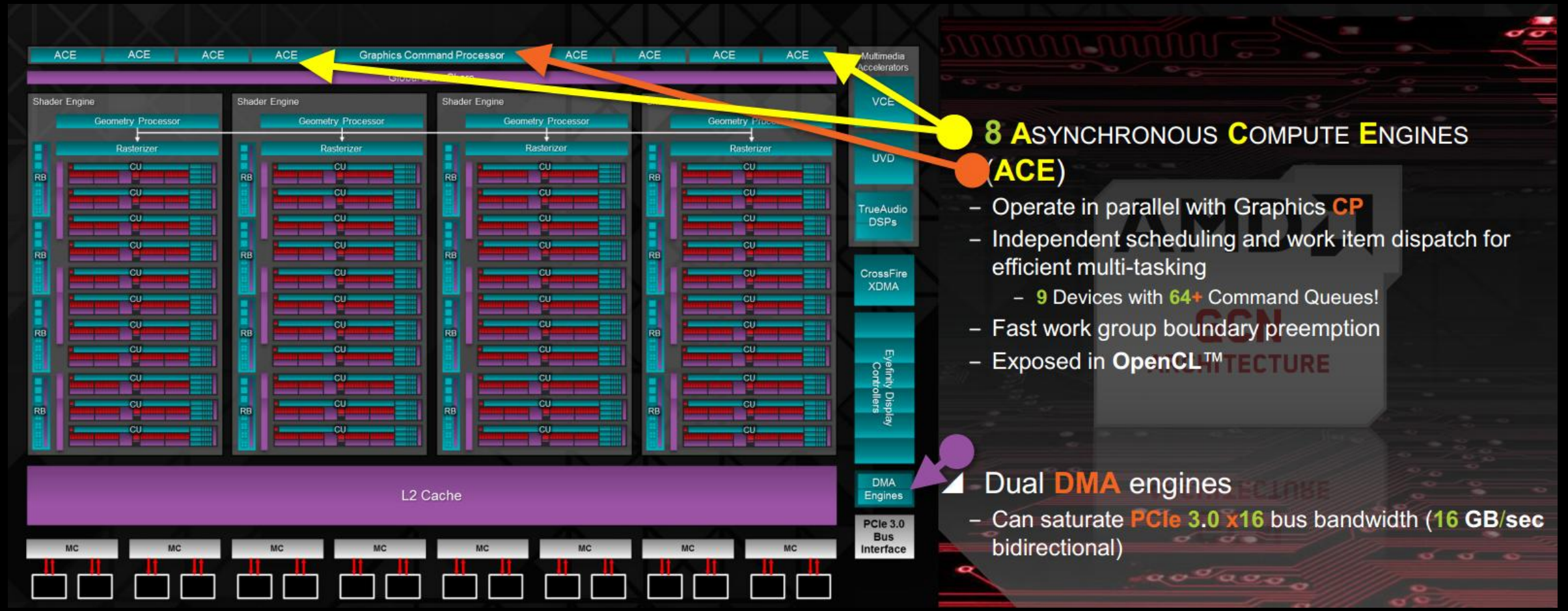
Read operation

method	0.5MB	1MB	10MB	100MB
Pre-pinned	5.2GB/s	7.4GB/s	10.9GB/s	12.GB/s
classic	2.GB/s	3GB/s	6.GB/s	6.5GB/s

- **Pre-pinned path is supported for the following calls**
  - clEnqueueRead/WriteBuffer
  - clEnqueueRead/WriteImage
  - clEnqueueRead/WriteBufferRect
- **CL image calls must use pre-pinned mapped buffers on the host side**

DATA TRANSFER

PARALLELISM



DATA TRANSFER PARALLELISM

AMD

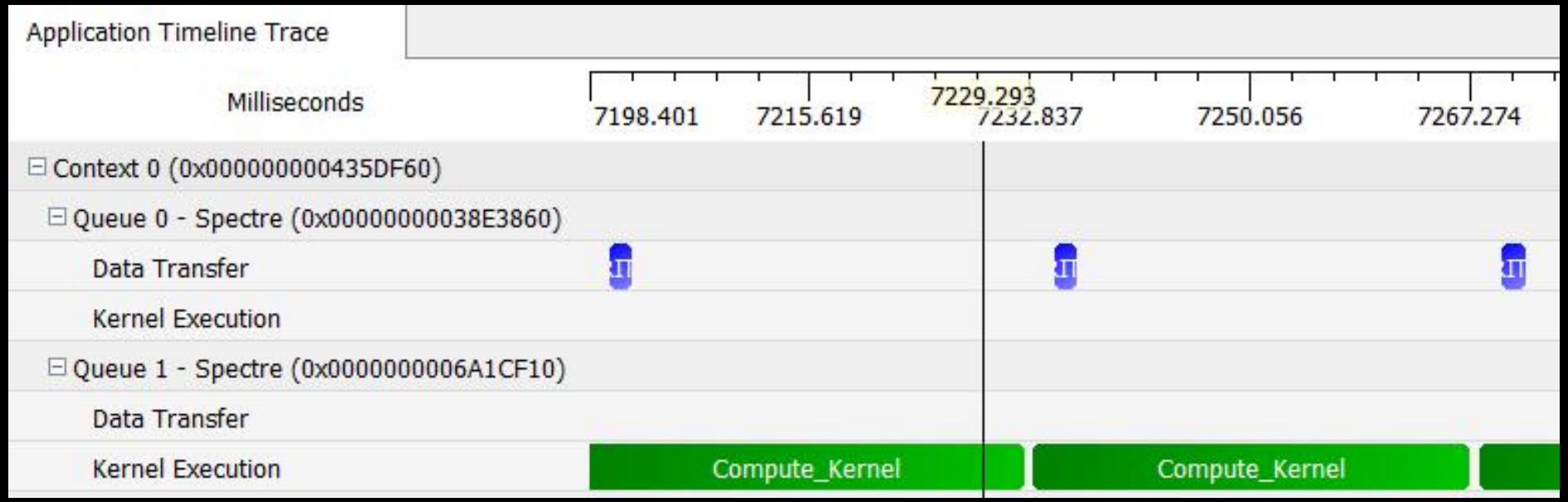
- **Modern GPUs have 2 DMAs engines**
  - Can do read/write operation in parallel
  - Can also overlap compute and transfer
- **To achieve parallel compute and transfer in OpenCL, one need to use multiple queues**

```
QueueRead = clCreateCommandQueue()
QueueWrite = clCreateCommandQueue()
QueueCompute1 = clCreateCommandQueue();
clEnqueueReadBuffer(QueueRead )
clEnqueueWriteBuffer(QueueWrite )
clEnqueueNDRangeKernel(QueueCompute1 )

clFlush(QueueRead);….
```

- **On our OpenCL runtime, odd queue number are allocated to DMA1, even queue number to DMA2. Be careful about the order your create your queues**

DATA TRANSFER PARALLELISM



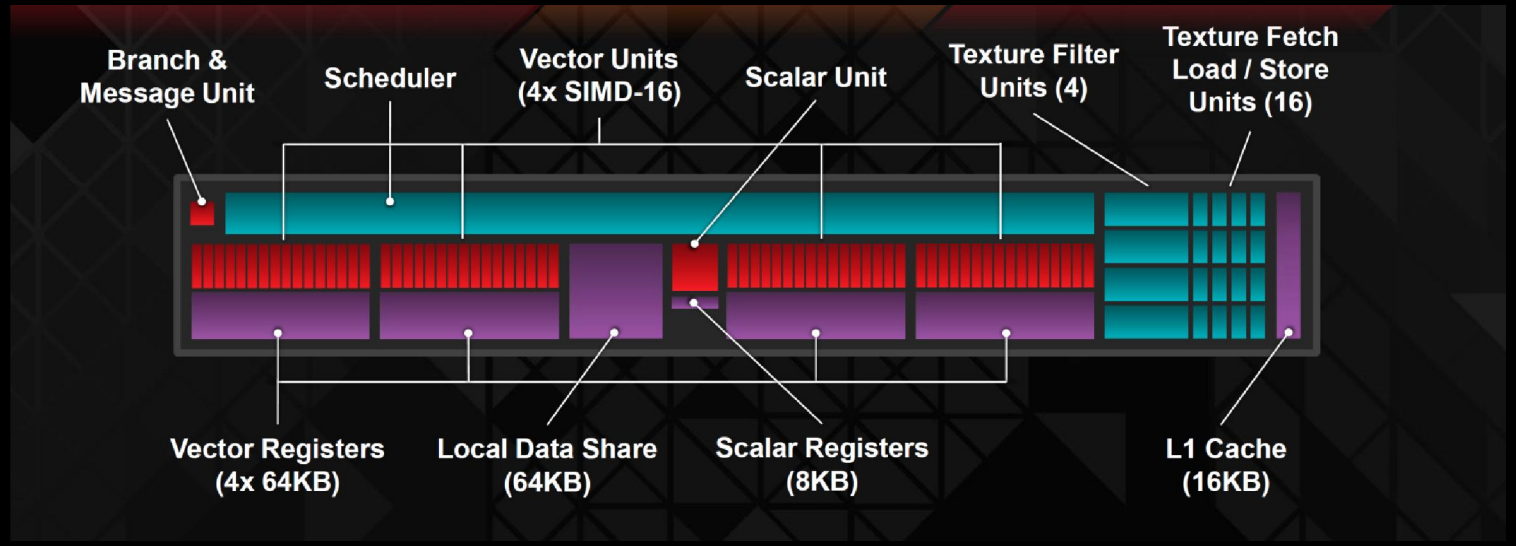
DATA TRANSFER

- http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, chapter 5.6.2, page 89
- https://github.com/AMD-FirePro?tab=repositories

PARALLELISM

AMD

- **OpenCL performance comes from parallelism**
  - clEnqueueNDRangeKernel (queue, kernel, dim, NULL, **globalsize** ,…)
  - You want to have the biggest global size as possible to spawn as many threads as possible on a massively parallel device
  - Hawaii (W9100/S9150) is composed of 44 CUs, each CU has 4 16-length SIMD => 2816 threads
  - This GPU can actually have 112640 active threads running at the same time.



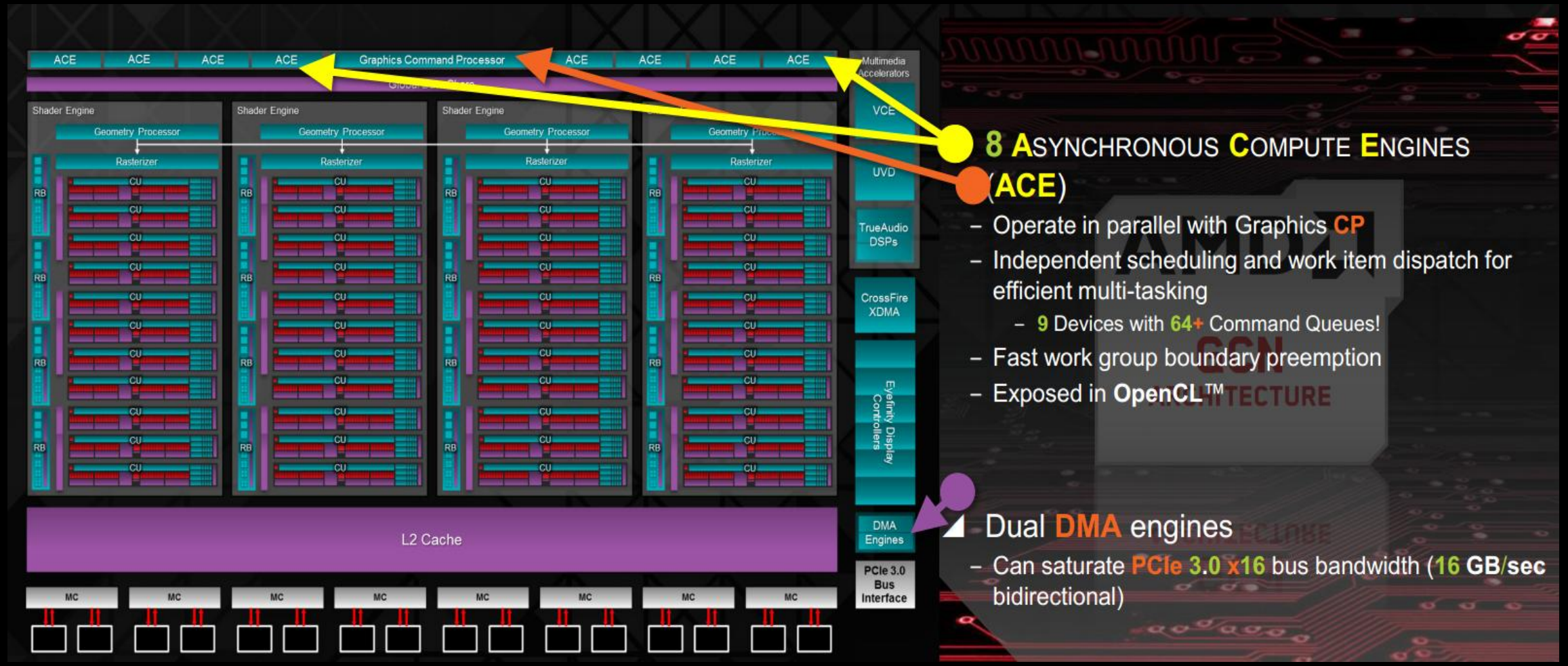
PARALLELISM

- A “hardware thread” (a wavefront) is composed of 64 threads
- A wavefront runs on one SIMD inside a CU => a wavefront executes in 4 steps
- Each SIMD can have 10 active wavefronts
- This means we can have 44*4*10*64 = 112640 active threads

Work-group size	Number of VGPRs	Number of SGPRs	LDS size

Variable	Value	Device Limit
Device Info		
Device name	Hawaii	
Number of compute units	40	
Max number of waves per compute unit	40	
Max number of work-groups per compute unit	16	
Wavefront size	64	
Kernel Info		
Kernel name	init_kernel	
Vector GPR usage per work-item	10	256
Scalar GPR usage per work-item	12	104
LDS usage per work-group	0	65536
Flattened work-group size	256	256
Flattened global work size	256	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	40	40
Number of waves limited by Scalar GPR and Work-group size	40	40
Number of waves limited by LDS and Work-group size	40	40
Number of waves limited by Work-group size	40	40
Limiting factor(s)	None	
Estimated occupancy	100%	

PARALLELISM



PARALLELISM
BINARY SEARCH

AMD

- This is an example from the SDK used to show a CL2.0 feature
- Though I don’t think it always exposes the fastest way of doing a search in an array
- In this example, we do a N-search where N = 256 => at each steps we have M/256 threads running on the GPU. Where M is the array size. This is not always enough to fill the GPU
- By having each thread looking into a different entry in the array (one thread per entry) we can increase the parallelism and actually write a simpler kernel when the array in not too big

Performance	4096	262144	4194304	16777216
M-search	0.01	0.03	0.3	2.23
256-search	0.08	0.08	0.3	1.1

NB threads	4096	262144	4194304	16777216
M-search	4096	262144	4194304	16777216
256-search	256	1024	16384	65536

PARALLELISM

AMD

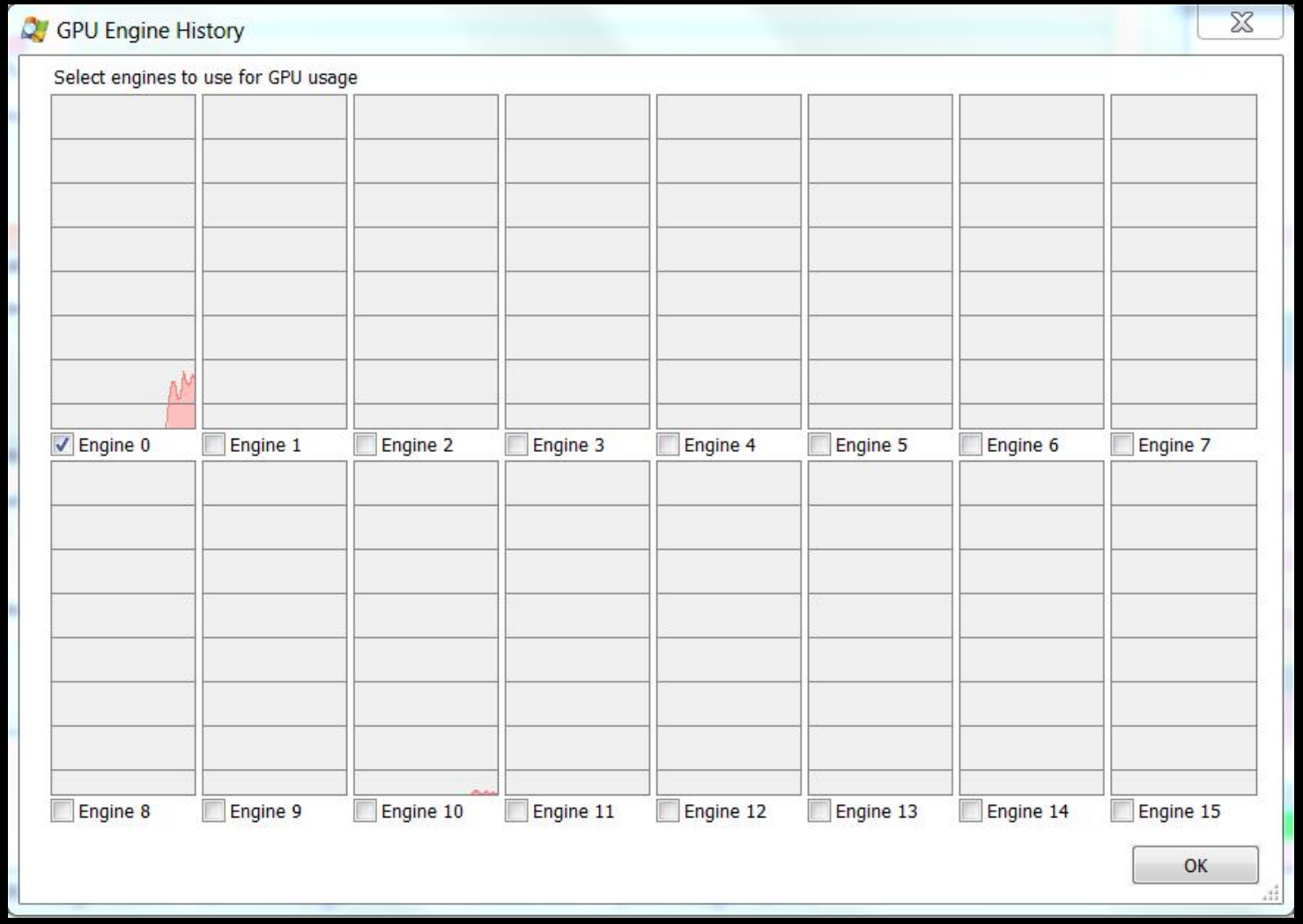
- What is important to notice is we need parallelism to use the GPU.
- If we don’t have a big array, the algorithm exposing the more parallelism will give the best performance.
- Though a brut force approach may not be useful when the array is really big and a N-search can be used to reduce the size before reapplying our brute force algorithm.

PARALLELISM
MULTI-TASKING

- **Sometimes you have a lot of small independent batches to process**
  - Big linear systems break down in small pieces
  - Several small meshes to animate
  - ….
- **Our GPUs have 8 ACEs, Asynchronous Compute Engines.**
  - ACEs are responsible for compute shader scheduling
  - ACEs are independent
  - ACEs dispatch tasks to the compute engines as resources permit
- **ACEs are independent virtual engine, enabling true Multiprocessor operation.**
- **In OpenCL you can access them by using multiple OpenCL command queues!!**

PARALLELISM

GPU ENGINES EXPOSED TO OS, PROCESS EXPLORER



MEMORY ACCESS

AMD

- Coalesced memory access
  - Means adjacent thread access adjacent memory
- Our cache line is 64 bytes
  - any fetch request will actually fetch 64 bytes, even if you only look into a single char
- Data/algorithm needs to be arranged to maximize the bandwidth usage
  - For a simple vector addition a simple linear access is enough to have coalesced access

T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	…
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	…
B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]	…

- Here each memory fetch will be in the same cache line
  - &A[0] =0xNNNN, &A[1]=0xNNNN + 4, …, &A[15] = 0xNNNN + 60, &A[16] = 0xNNNN + 64
  - We maximize the bandwidth usage

MEMORY ACCESS

AMD

- What happened if we don’t have coalesced access.
- For example we have a stride between each relevant data we need to look at

T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	…
A[0]	A[1+16]	A[2+16]	A[3+16]	A[4+16]	A[5+16]	A[6+16]	A[7+16]	A[8+16]	A[9+16]	…
B[0]	B[1+16]	B[2+16]	B[3+16]	B[4+16]	B[5+16]	B[6+16]	B[7+16]	B[8+16]	B[9+16]	…

- Here each fetch will use one cache line
  - &A[0]=0xNNNN, &A[17] =0xNNNN + 68…
- For the first 10 threads we will fetch 2*64*10 = 1280 bytes….
- ….while only 80 bytes are useful, we waste nearly 1kB of data and we only use 1/16 of the bandwidth
- Here we have a simple test case. But this needs to be taken into account when working on more complex data. This is why SoA needs to be preferred over AoS.
- Coherency and locality of your data are key to achieve the best performance

MEMORY ACCESS

GLOBAL_MEMORY_BANDWIDTH

Read linear uncached	Read Linear cached	Read Single cache line	Read random	Read uncombine uncached
325 GB/s	1473 GB/s	3825GB/s	54GB/s	182GB/s

//read linear cached

```
val = val + input[gid + 0];
val = val + input[gid + 1]; // this is in l1 cached as requested from previous fetch
val = val + input[gid + 2]; // this is in l1 cached…
…
output[gid] = val;
```

//read linear uncombined uncached

```
#define NUM_READS 32
val = val + input[gid * NUM_READS + 0];
 val = val + input[gid * NUM_READS + 1];
 val = val + input[gid * NUM_READS + 2];…
```

BEST WORK GROUP SIZE

AMD

- It is important to think about the work group size as it is how you will map the different work-items on the hardware
- On AMD HW, a wavefront is 64 threads => the most efficient work group sizes have to be multiple of 64
  - Using 65 threads in a work group will require two wavefronts to execute and waste 63 lanes.
- This value can easily be queried in OpenCL using this API
  - clGetKernelWorkGroupInfo (**CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE**)
  - It is available since OpenCL 1.1 and can help you writing more generic code to support different HW vendor

- You can also specify the work group size at compile time using __atribute__ in your OpenCL C code
  - __attribute__((reqd_work_group_size(8,8,1))) __kernel void ... This will help the compiler and produce a code specific to a work group size of 8x8.
  - This can help for some optimizations.
  - For example, in such case our compiler won't generate barrier instruction for barrier(); but just a fence

OCCUPANCY

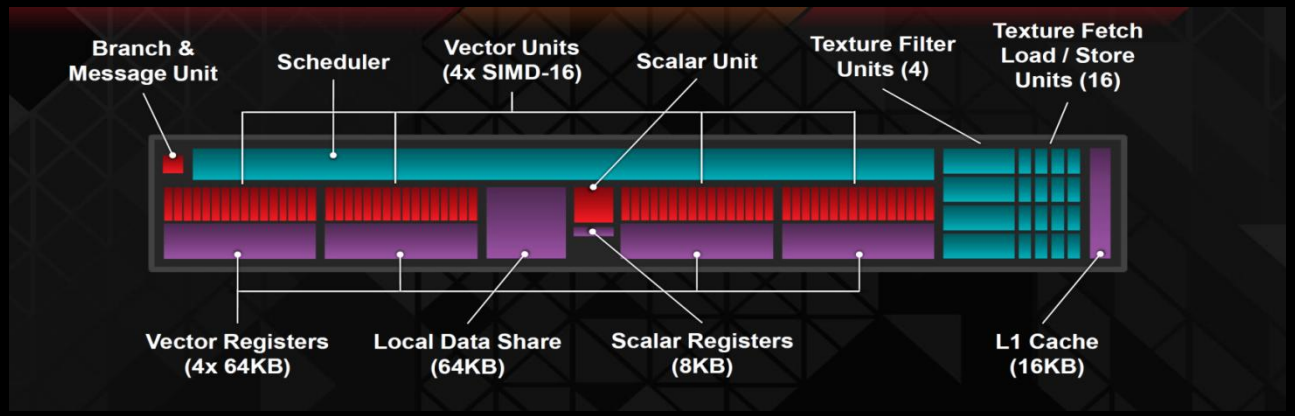
AMD

- This is the capacity to keep the GPU busy by being able to have several wavefronts running on the same SIMD
- One SIMD can have up to 10 active wavefronts
- Switching between wavefronts allows to avoid waiting for memory transaction. A fetch/write can take 100s of clock cycle to execute while an add instruction on float takes one clock.

OCCUPANCY

VGPRS

- We have 64 KB of registers (VGPRs) per SIMD. This is 32 bits registers



- This means 16384 VGRPs.
- A SIMD runs a wavefront of 64 threads => 256 VGPRs per thread maximum
- If a kernel uses more than 256 VGPRs, we start spilling which will affect greatly the performance as the spilling occurs in global memory
- If we use less than 256 VGPRs we can actually have several wavefront running on the same SIMD
  - With 128 VGPRs we can have 2 waves
  - With 25 we can have 10 waves

OCCUPANCY
SPILLING

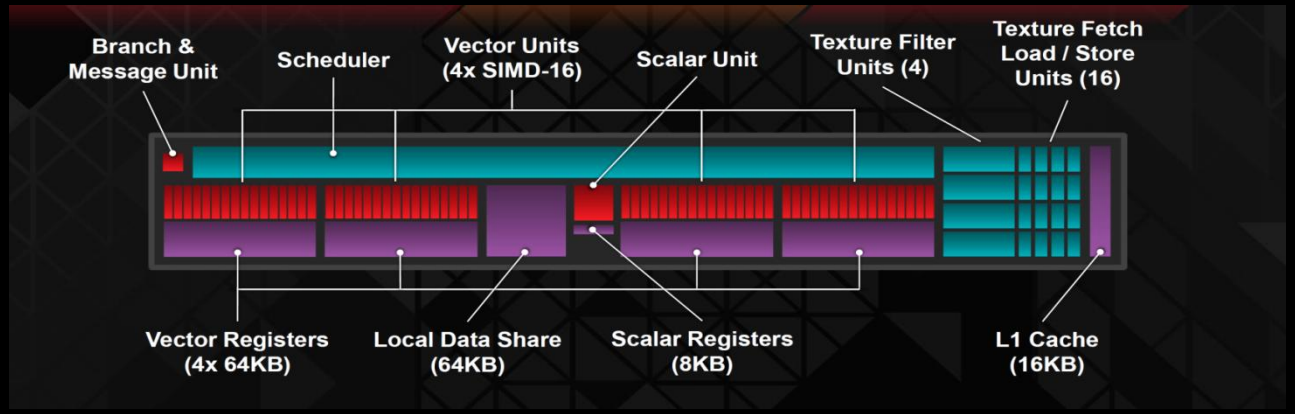
- There are two main reasons to spill
- A big kernel
  - The bigger the kernel is, the more registers you are likely to use
  - In that case the compiler will try to spill to improve the occupancy. But sometime, it is so big that we spill and have a low occupancy (ie megakernel for raytracer : 15k-22k lines for a single kernel)
  - The solutions are to “split” the kernel into smaller ones and/or change the algorithm
- Forcing an unroll, this will actually behave like a big kernel
  - Avoid unrolling if you find you use too many registers
- One can easily find the VGPRs usage.
  - codeXL, our profiling tool, shows the GPR usage and the spilling (scratch reg).
  - These information can be directly find in the isa code
  - You can access the isa code with codeXL or by dumping it
  - AMD_OCL_BUILD_OPTIONS_APPEND=-save-temps

```
extUserElementCount  = 0;
NumVgprs             = 11;
NumSgprs             = 20;
FloatMode            = 192;
IeeeMode             = 0;
FlatPtr32            = 0;
ScratchSize          = 0 dwords/thread;
LDSByteSize          = 0 bytes/workgroup (compile time only);
```

OCCUPANCY

LOCAL MEMORY/WORK GROUP SIZE

- We have 64 KB of LDS per CU, but only 32 KB available per workgroup



- If you have a work group of size 64 and require 32KB of LDS, you won't be able to run more than 2 waves per CU => very low occupancy as two SIMDs won't be used
- If you have a work group of size 256 and use 32 KB of LDS, you can have up to two waves per SIMD
  - You actually need 8KB per wavefront
  - Thus you can have up to 8 wavefronts running per CU => 2 per SIMD
- LDS is a very fast low latency programmable cache, but it is a limited amount of resource. Use it sparingly

OCCUPANCY

AMD

- Use codeXL to see where you lose in occupancy.
- The conjunction of LDS/work group size and registers usage will impact the occupancy

Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	40	40
Number of waves limited by Scalar GPR and Work-group size	40	40
Number of waves limited by LDS and Work-group size	40	40
Number of waves limited by Work-group size	40	40
Limiting factor(s)	None	
Estimated occupancy	100%	

- On very slow kernel the problem can be obvious, request of 32KB of LDS, using more than 128GPRs….
- Though if you are not memory bound having a low occupancy is not necessarily bad. But very few kernels don’t depend on the bandwidth

BRANCHING

AMD

- In a wavefront all the lanes will execute the same instructions
- In case of branching if one thread diverge in a wavefront we will need to go through both path and mask the result of the unwanted path for the others threads

```
if(get_local_id(0)%2==0)
{}   //executes in T1
else
{}   //executes in T2
```

- Here the overall time is T1+T2
- When branching can be avoided, you will achieve better performance
- In the example above consider rearranging the data or having two different kernels