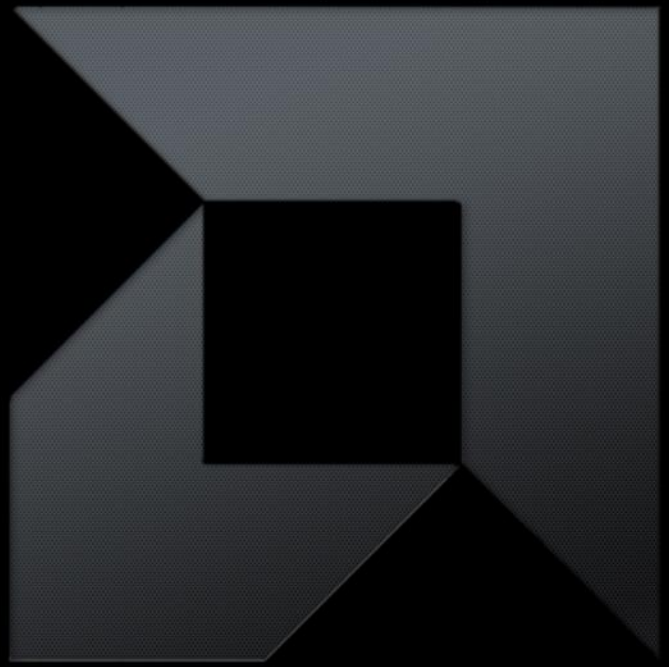


HSA AND THE MODERN GPU

Lee Howes
AMD Heterogeneous System Software

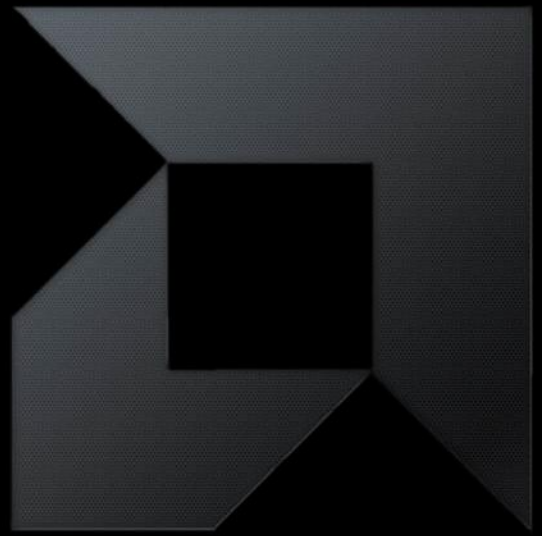
AMD

HSA AND THE MODERN GPU

- In this brief talk we will cover three topics:
  - Changes to the shader core and memory system
  - Changes to the use of pointers
  - Architected definitions to use these new features

- We'll look both at how the hardware is becoming more flexible, and how the changes will benefit OpenCL implementations.

THE HD7970 AND GRAPHICS CORE NEXT



AMD

GPU EXECUTION AS WAS

- We often view GPU programming as a set of independent threads, more reasonably known as “work items” in OpenCL:

```
kernel void blah(global float *input, global float *output) {
  output[get_global_id(0)] = input[get_global_id(0)];
}
```

- Which we flatten to an intermediate language known as AMD IL:

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

- Note that AMD IL contains short vector instructions

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads

- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
  – Each work item describes a lane of execution
  – Multiple work items execute together in SIMD fashion with a single program counter
  – Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads

- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

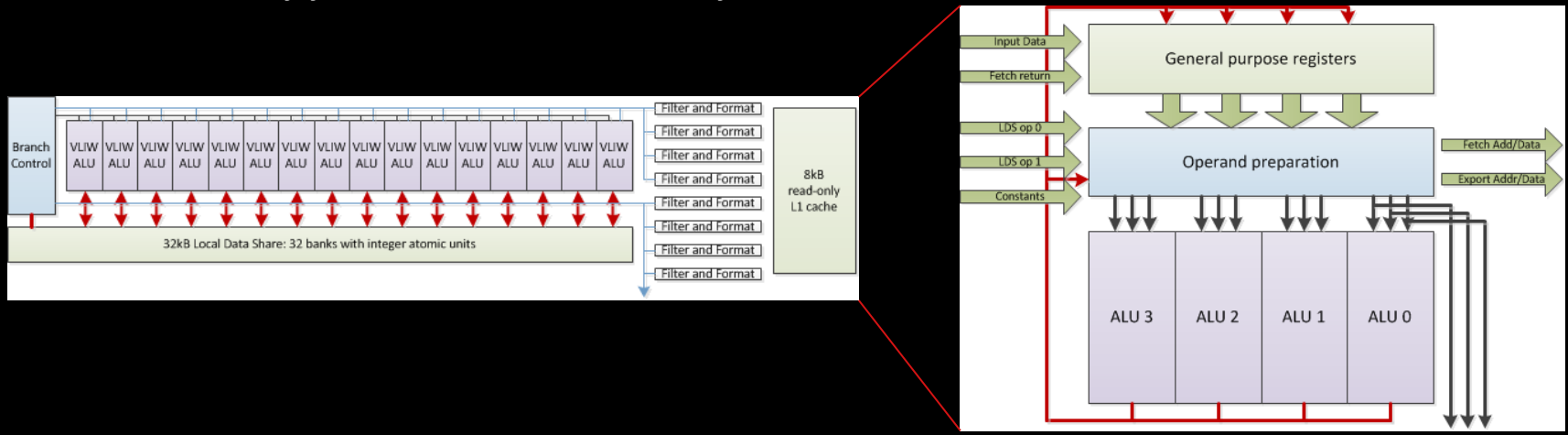

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```



```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

IT WAS NEVER QUITE THAT SIMPLE

- The HD6970 architecture and its predecessors were combined multicore SIMD/VLIW machines
  - Data-parallel through hardware vectorization
  - Instruction parallel through both multiple cores and VLIW units
- The HD6970 issued a 4-way VLIW instruction per work item
  - Architecturally you could view that as a 4-way VLIW instruction issue per SIMD lane
  - Alternatively you could view it as a 4-way VLIW issue of SIMD instructions



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
     0  w: LSHL       ____,  R0.x,  2
     1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
     2  y: LSHR       R0.y,  PV1.z,  2
     3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
        y: MULLO_INT  ____,  R1.x,  KC1[1].x
        z: MULLO_INT  ____,  R1.x,  KC1[1].x
        w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
     4  VFETCH R2.x___, R0.y, fc153
        FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
     5  w: ADD_INT    ____,  R0.x,  R1.x
     6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
     7  y: LSHL       ____,  PV6.z,  2
     8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
     9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
      0  w: LSHL       ____,  R0.x,  2
      1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
      2  y: LSHR       R0.y,  PV1.z,  2
      3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
         y: MULLO_INT  ____,  R1.x,  KC1[1].x
         z: MULLO_INT  ____,  R1.x,  KC1[1].x
         w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
      4  VFETCH R2.x___, R0.y, fc153
         FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
      5  w: ADD_INT    ____,  R0.x,  R1.x
      6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
      7  y: LSHL       ____,  PV6.z,  2
      8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
      9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

Clause header

Work executed by the shared scalar unit

WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
     0  w: LSHL       ____,  R0.x,  2
     1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
     2  y: LSHR       R0.y,  PV1.z,  2
     3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
        y: MULLO_INT  ____,  R1.x,  KC1[1].x
        z: MULLO_INT  ____,  R1.x,  KC1[1].x
        w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
     4  VFETCH R2.x___, R0.y, fc153
        FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
     5  w: ADD_INT    ____,  R0.x,  R1.x
     6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
     7  y: LSHL       ____,  PV6.z,  2
     8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
     9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

Clause header
Work executed by the shared scalar unit

Clause body
Units of work dispatched by the shared scalar unit

WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
     0  w: LSHL       ____,  R0.x,  2
     1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
     2  y: LSHR       R0.y,  PV1.z,  2
     3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
        y: MULLO_INT  ____,  R1.x,  KC1[1].x
        z: MULLO_INT  ____,  R1.x,  KC1[1].x
        w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
     4  VFETCH R2.x___, R0.y, fc153
        FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
     5  w: ADD_INT    ____,  R0.x,  R1.x
     6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
     7  y: LSHL       ____,  PV6.z,  2
     8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
     9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

Clause header
Work executed by the shared scalar unit

VLIW instruction packet
Compiler-generated instruction level parallelism for the VLIW unit.
Each instruction (x, y, z, w) executed across the vector.

Clause body
Units of work dispatched by the shared scalar unit

WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
     0  w: LSHL       ____,  R0.x,  2
     1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
     2  y: LSHR       R0.y,  PV1.z,  2
     3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
        y: MULLO_INT  ____,  R1.x,  KC1[1].x
        z: MULLO_INT  ____,  R1.x,  KC1[1].x
        w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
     4  VFETCH R2.x___, R0.y, fc153
        FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
     5  w: ADD_INT    ____,  R0.x,  R1.x
     6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
     7  y: LSHL       ____,  PV6.z,  2
     8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
     9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

Clause header
Work executed by the shared scalar unit

VLIW instruction packet
Compiler-generated instruction level parallelism for the VLIW unit.
Each instruction (x, y, z, w) executed across the vector.

Clause body
Units of work dispatched by the shared scalar unit

Notice the poor occupancy of VLIW slots

WHY DID WE SEE INEFFICIENCY?

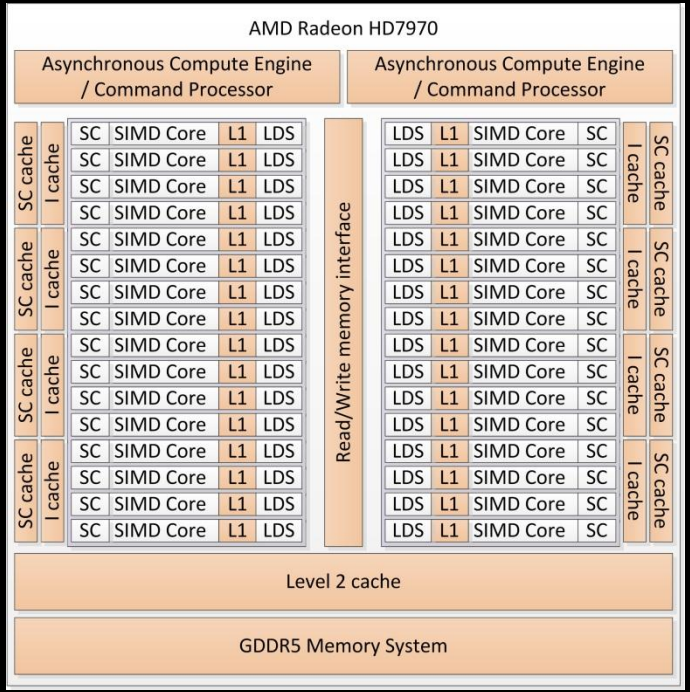
- The architecture was well suited to graphics workloads:
  - VLIW was easily filled by the vector-heavy graphics kernels
  - Minimal control flow meant that the monolithic, shared thread scheduler was relatively efficient

- Unfortunately, workloads change with time.

- So how did we change the architecture to improve the situation?

AMD RADEON HD7970 - GLOBALLY

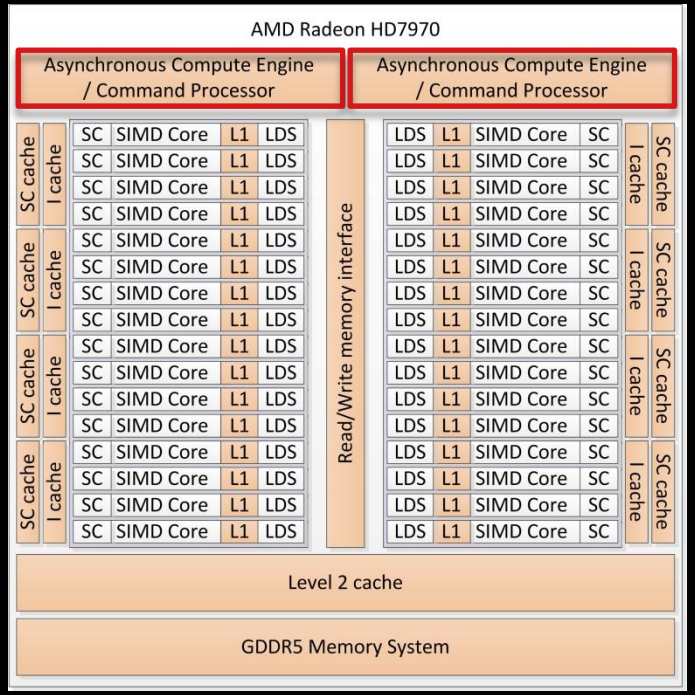
- Brand new – but at this level it doesn't look too different



AMD RADEON HD7970 - GLOBALLY

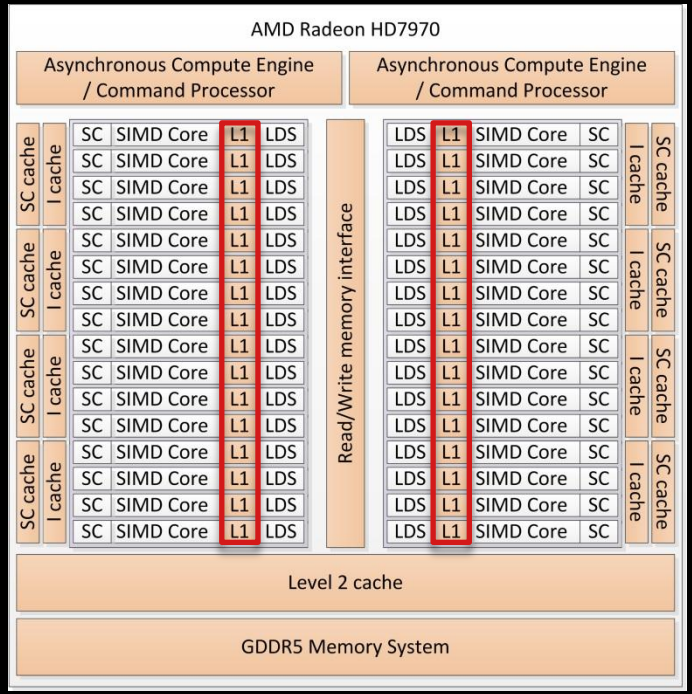
- Brand new – but at this level it doesn't look too different

- Two command processors
  - Capable of processing two command queues concurrently



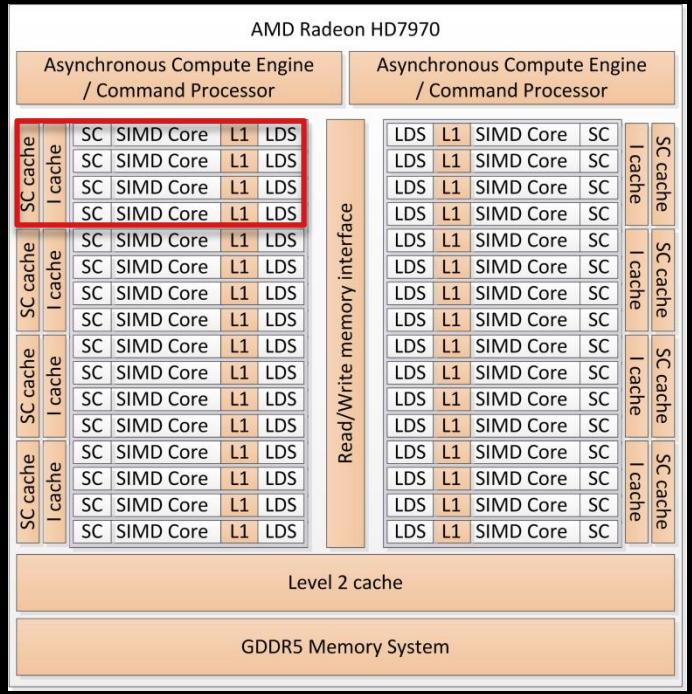
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
  - Capable of processing two command queues concurrently
- Full read/write L1 data caches
- SIMD cores grouped in fours
  - Scalar data and instruction cache per cluster
  - L1, LDS and scalar processor per core
- Up to 32 cores / compute units



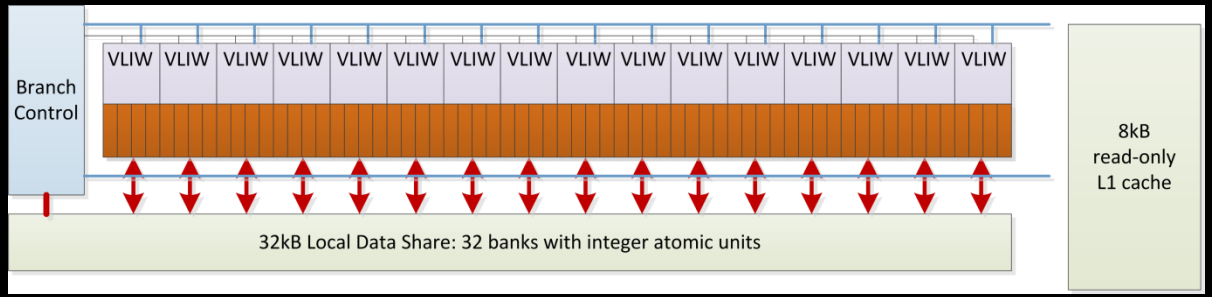
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
  - Capable of processing two command queues concurrently
- Full read/write L1 data caches
- SIMD cores grouped in fours
  - Scalar data and instruction cache per cluster
  - L1, LDS and scalar processor per core
- Up to 32 cores / compute units



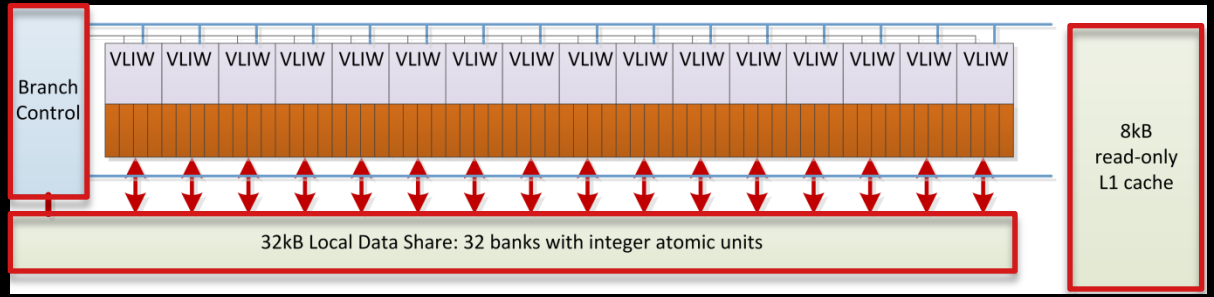
THE SIMD CORE

- The SIMD unit on the HD6970 architecture had a branch control but full scalar execution was performed globally



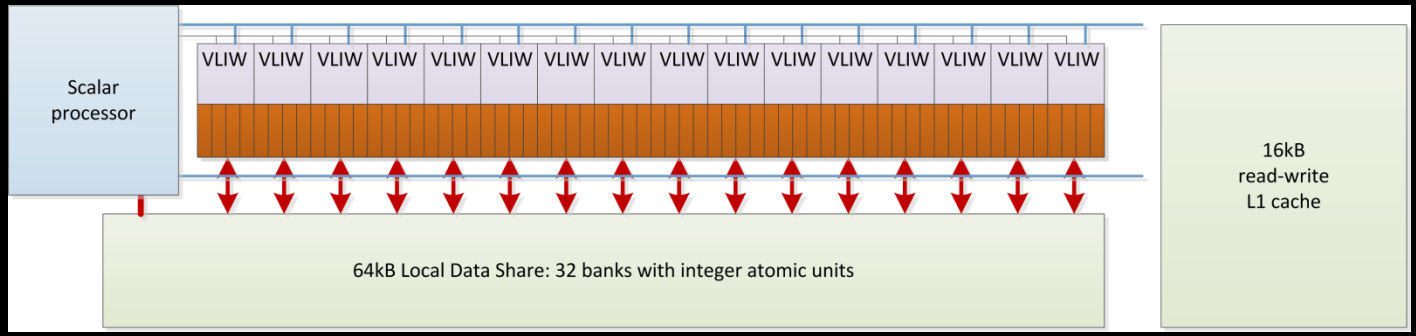
THE SIMD CORE

- The SIMD unit on the HD6970 architecture had a branch control but full scalar execution was performed globally



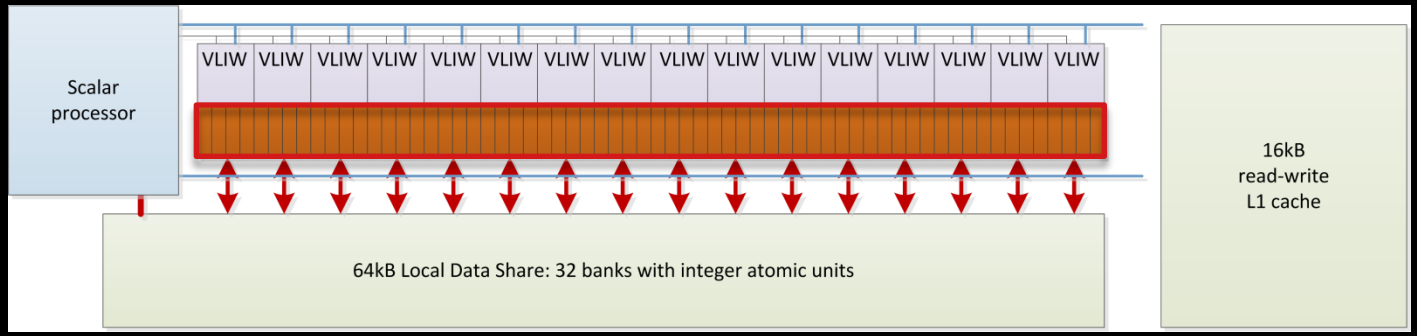
THE SIMD CORE

- On the HD7970 we have a full scalar processor and the L1 cache and LDS have been doubled in size



THE SIMD CORE

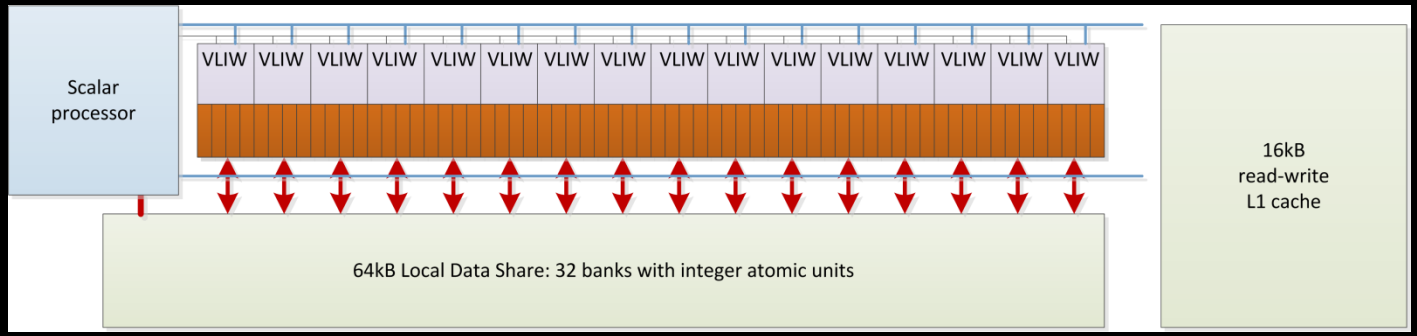
- On the HD7970 we have a full scalar processor and the L1 cache and LDS have been doubled in size
- Then let us consider the VLIW ALUs



AMD

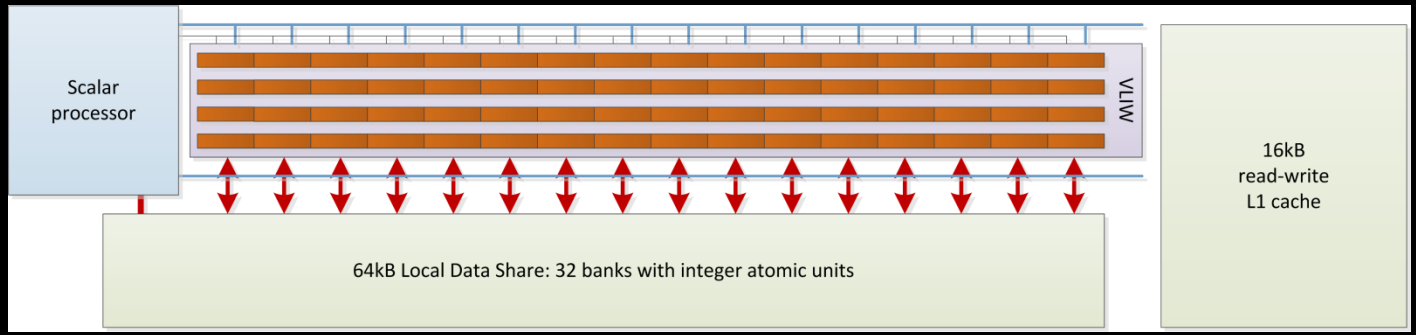
THE SIMD CORE

- Remember we could view the architecture two ways:
  - An array of VLIW units



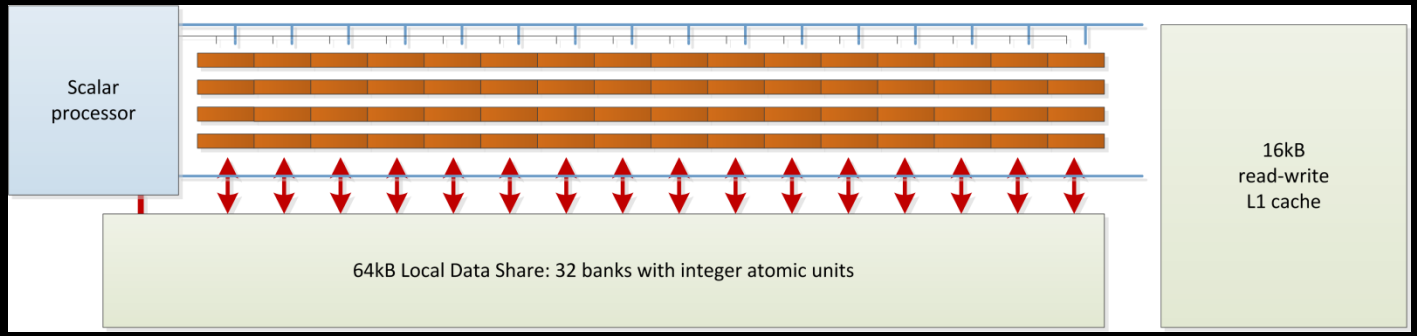
THE SIMD CORE

- Remember we could view the architecture two ways:
  - An array of VLIW units
  - A VLIW cluster of vector units



THE SIMD CORE

- Now that we have a scalar processor we can dynamically schedule instructions rather than relying on the compiler
- No VLIW!



- The heart of Graphics Core Next:
  - A scalar processor with four 16-wide vector units
  - Each lane of the vector, and hence each IL work item, is now scalar

THE NEW ISA AND BRANCHING

- Simpler and more efficient
- Instructions for both sets of execution units inline
- No VLIW
  - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recursion

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

v_cmp_gt_f32     r0,r1          //a > b, establish VCC
s_mov_b64        s0,exec        //Save current exec mask
s_and_b64        exec,vcc,exec  //Do "if"
s_cbranch_vccz   label0         //Branch if all lanes fail
v_sub_f32        r2,r0,r1       //result = a - b
v_mul_f32        r2,r2,r0       //result=result * a

    :
s_andn2_b64      exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz  label1         //Branch if all lanes fail
v_sub_f32        r2,r1,r0       //result = b - a
v_mul_f32        r2,r2,r1       //result = result * b

s_mov_b64        exec,s0        //Restore exec mask
```

THE NEW ISA AND BRANCHING

- Simpler and more efficient
- Instructions for both sets of execution units inline
- No VLIW
  - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recursion

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

v_cmp_gt_f32      r0,r1          //a > b, establish VCC
s_mov_b64         s0,exec        //Save current exec mask
s_and_b64         exec,vcc,exec  //Do "if"
s_cbranch_vccz    label0         //Branch if all lanes fail
v_sub_f32         r2,r0,r1       //result = a - b
v_mul_f32         r2,r2,r0       //result=result * a

    :
s_andn2_b64       exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz   label1         //Branch if all lanes fail
v_sub_f32         r2,r1,r0       //result = b - a
v_mul_f32         r2,r2,r1       //result = result * b

s_mov_b64         exec,s0        //Restore exec mask
```

THE NEW ISA AND BRANCHING

- Simpler and more efficient
- Instructions for both sets of execution units inline
- No VLIW
  - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recursion

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

v_cmp_gt_f32      r0,r1          //a > b, establish VCC
s_mov_b64         s0,exec        //Save current exec mask
s_and_b64         exec,vcc,exec  //Do "if"
s_cbranch_vccz    label0         //Branch if all lanes fail
v_sub_f32         r2,r0,r1       //result = a - b
v_mul_f32         r2,r2,r0       //result=result * a

    :
s_andn2_b64       exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz   label1         //Branch if all lanes fail
v_sub_f32         r2,r1,r0       //result = b - a
v_mul_f32         r2,r2,r1       //result = result * b

s_mov_b64         exec,s0        //Restore exec mask
```

THE NEW ISA AND BRANCHING

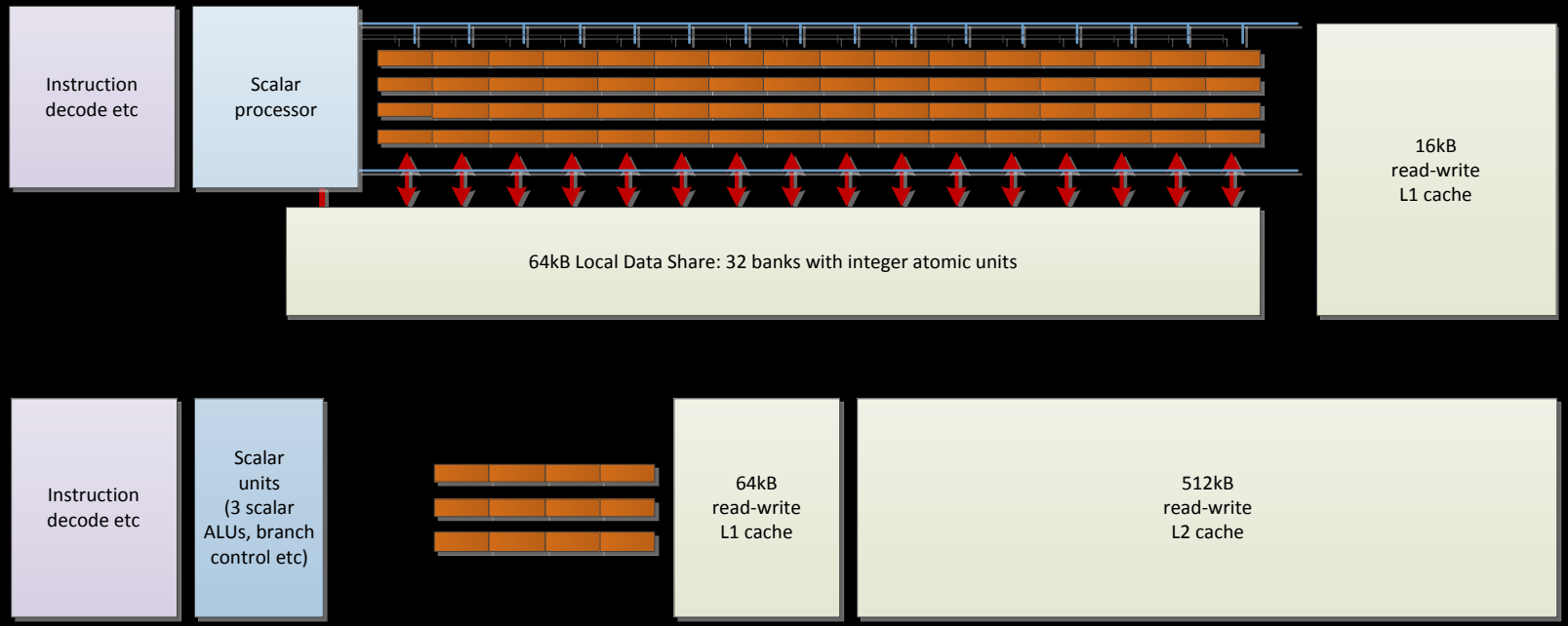
- Simpler and more efficient
- Instructions for both sets of execution units inline
- No VLIW
  - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recursion

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

v_cmp_gt_f32     r0,r1          //a > b, establish VCC
s_mov_b64        s0,exec        //Save current exec mask
s_and_b64        exec,vcc,exec  //Do "if"
s_cbranch_vccz   label0         //Branch if all lanes fail
v_sub_f32        r2,r0,r1       //result = a - b
v_mul_f32        r2,r2,r0       //result=result * a

    :
s_andn2_b64      exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz  label1         //Branch if all lanes fail
v_sub_f32        r2,r1,r0       //result = b - a
v_mul_f32        r2,r2,r1       //result = result * b

s_mov_b64        exec,s0        //Restore exec mask
```

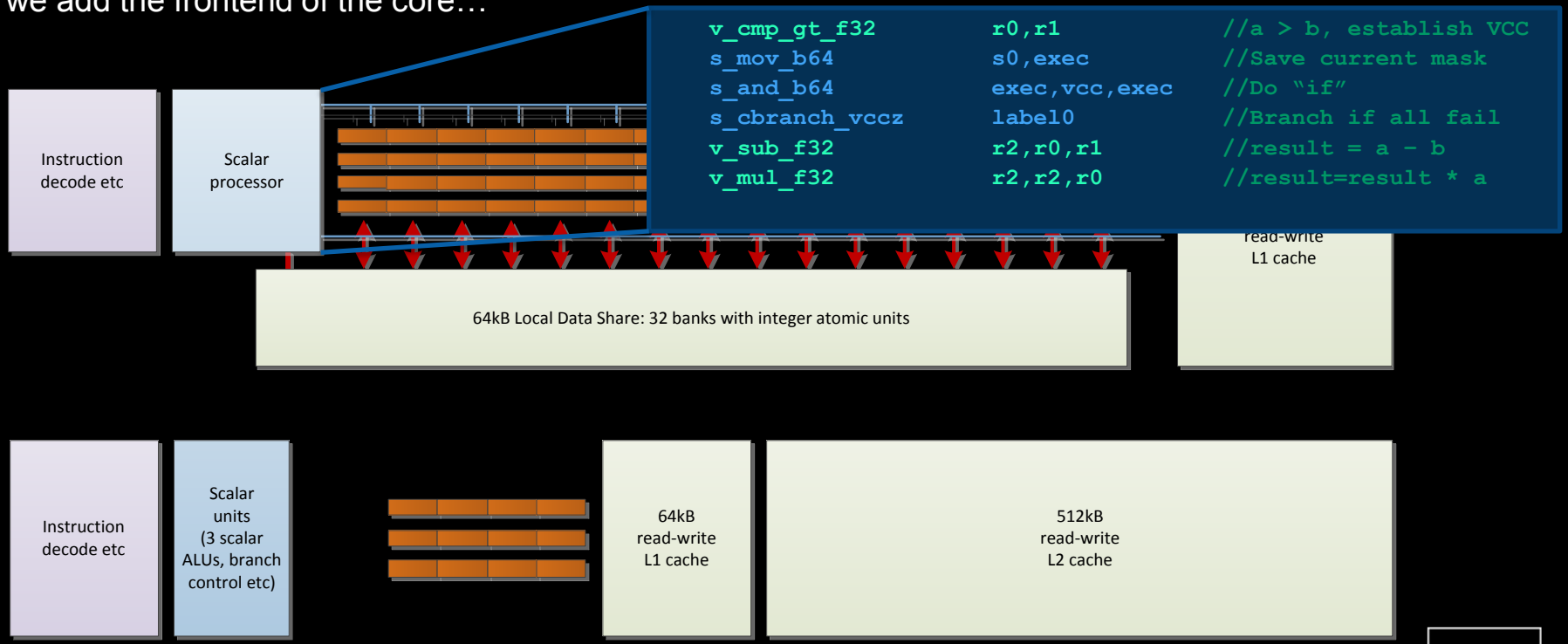
FAMILIAR?

- If we add the frontend of the core…



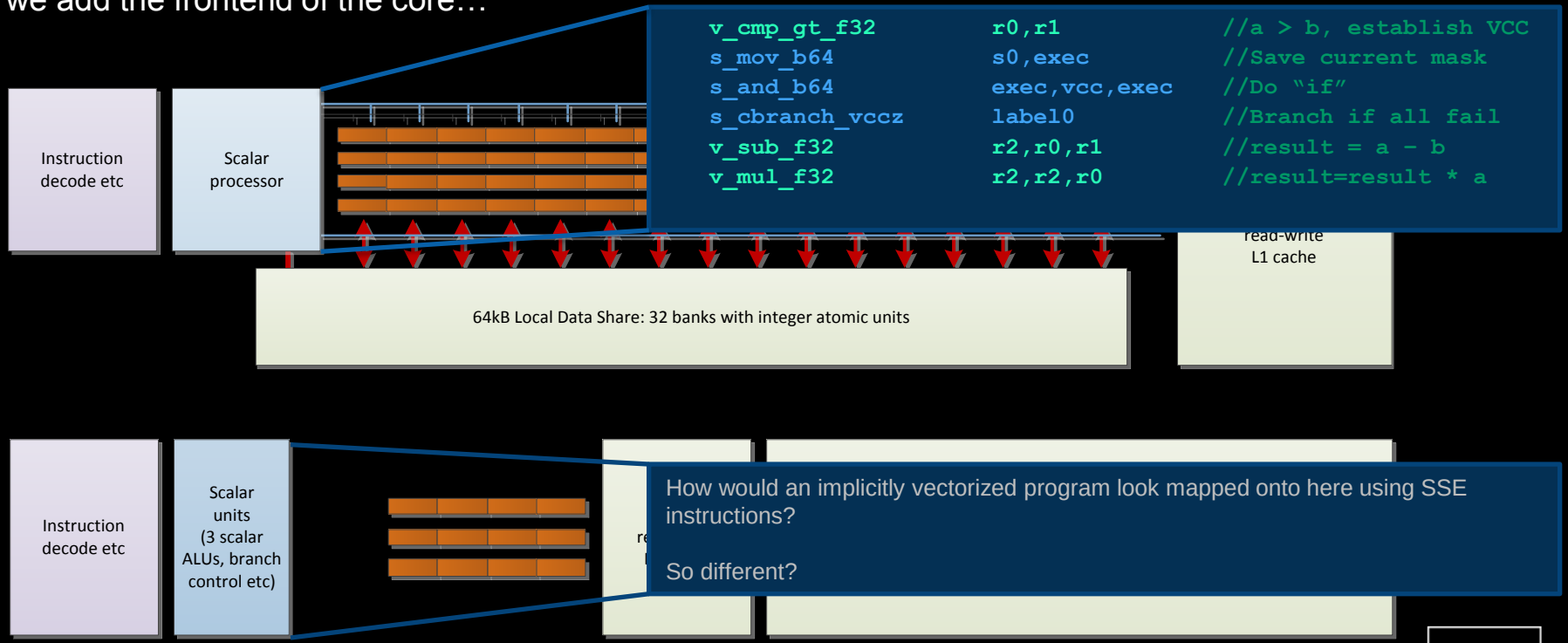
FAMILIAR?

- If we add the frontend of the core…



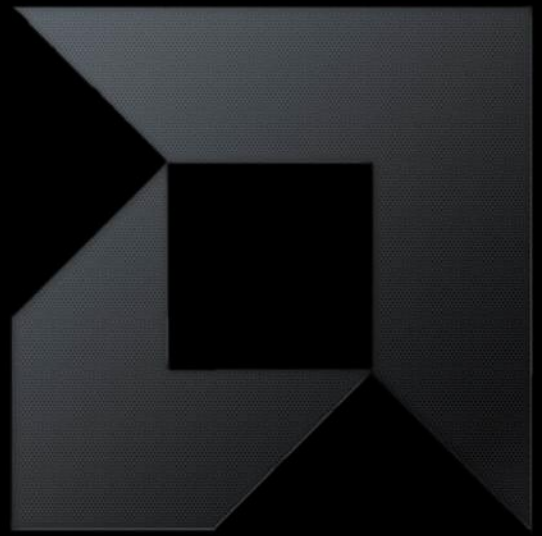
FAMILIAR?

- If we add the frontend of the core…



AMD

SHARED VIRTUAL MEMORY

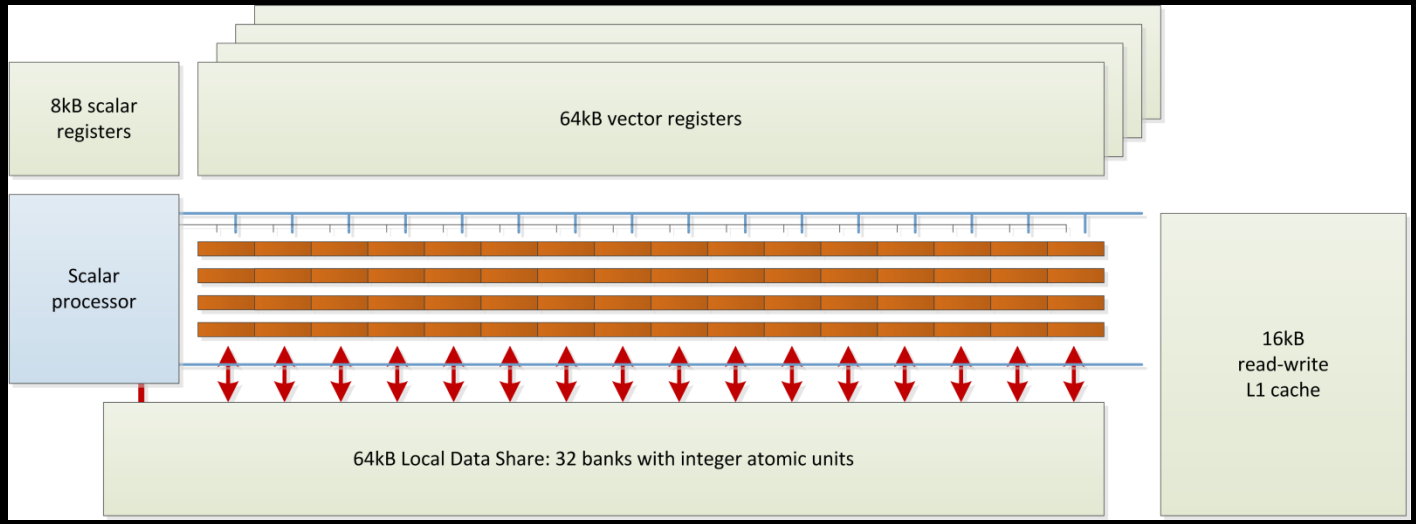


AMD

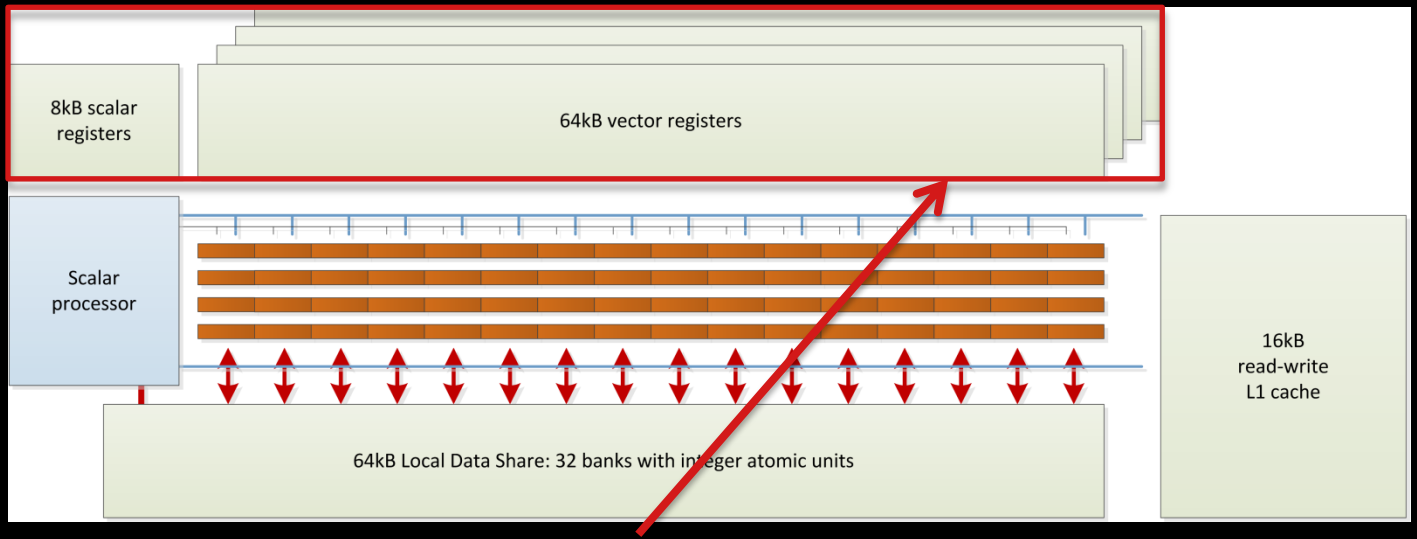
TAKING THE MEMORY SYSTEM CHANGES FURTHER AFIELD

- GCN-based devices are more flexible
  - Start to look like CPUs with few obvious shortcomings
- The read/write cache is a good start at improving the memory system
  - Improves efficiency on the device
  - Provides a buffer for imperfectly written code
- We needed to go a step further on an SoC
  - Memory in those caches should be the same memory used by the “host” CPU
  - In the long run, the CPU and GPU become peers, rather than having a host/slave relationship

WHAT DOES THIS MEAN?

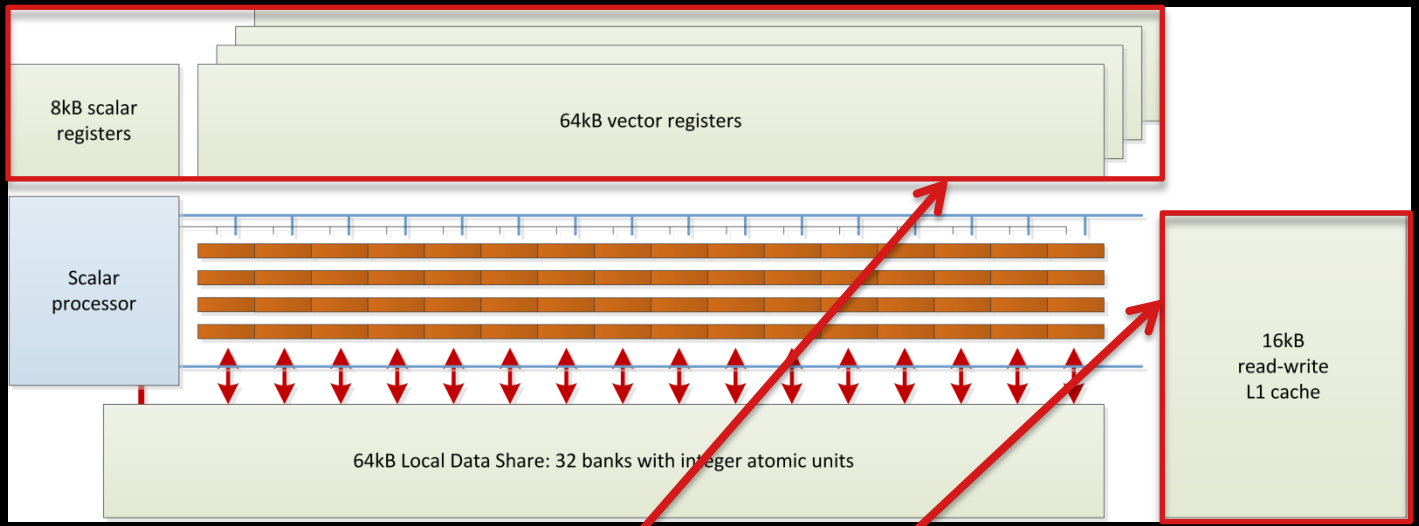


WHAT DOES THIS MEAN?



We can store x86 virtual pointers here

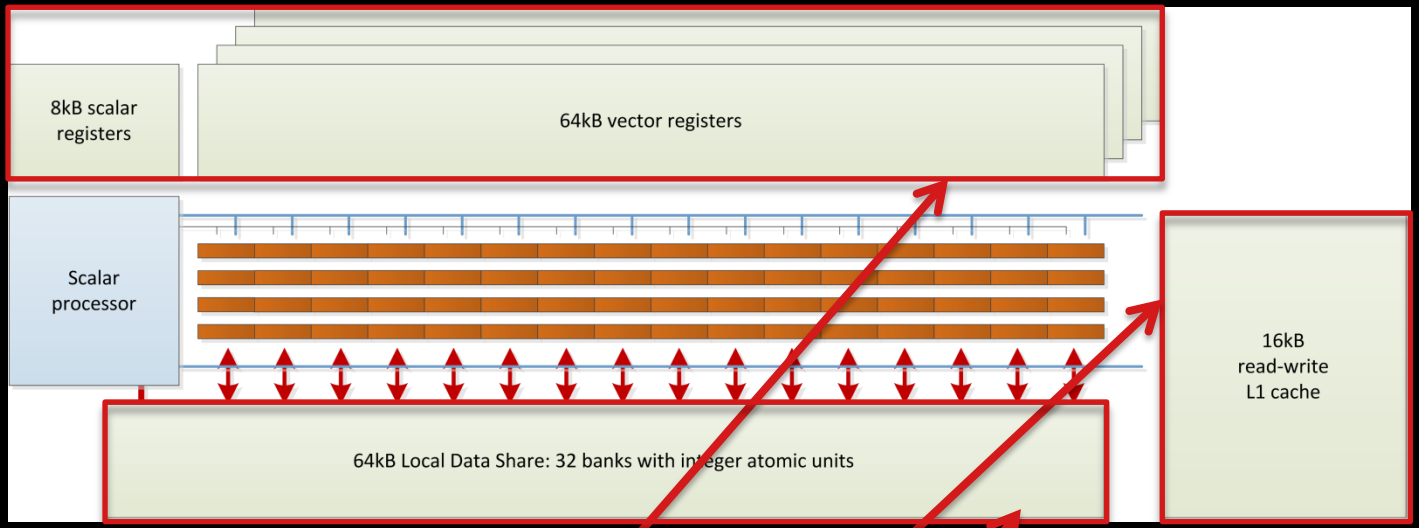
WHAT DOES THIS MEAN?



We can store x86 virtual pointers here

Data stored here is addressed in the same way as that on the CPU

WHAT DOES THIS MEAN?

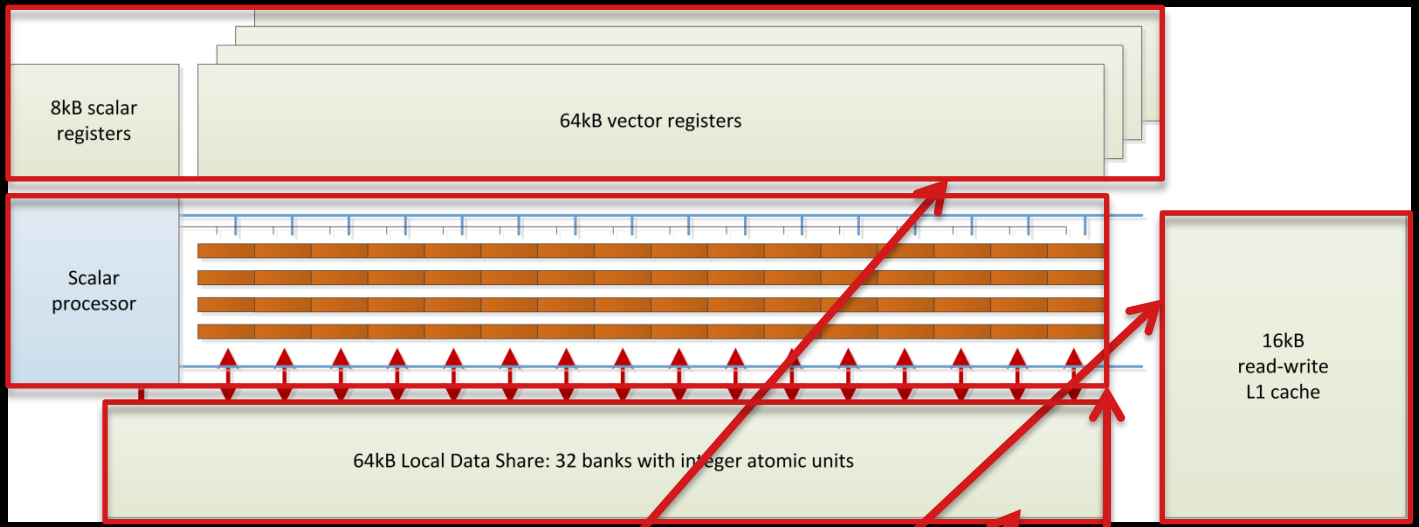


We can store x86 virtual pointers here

Data stored here is addressed in the same way as that on the CPU

We can map this into the same virtual address space

WHAT DOES THIS MEAN?



We can store x86 virtual pointers here

Data stored here is addressed in the same way as that on the CPU

We can map this into the same virtual address space

We can perform work on CPU data directly here

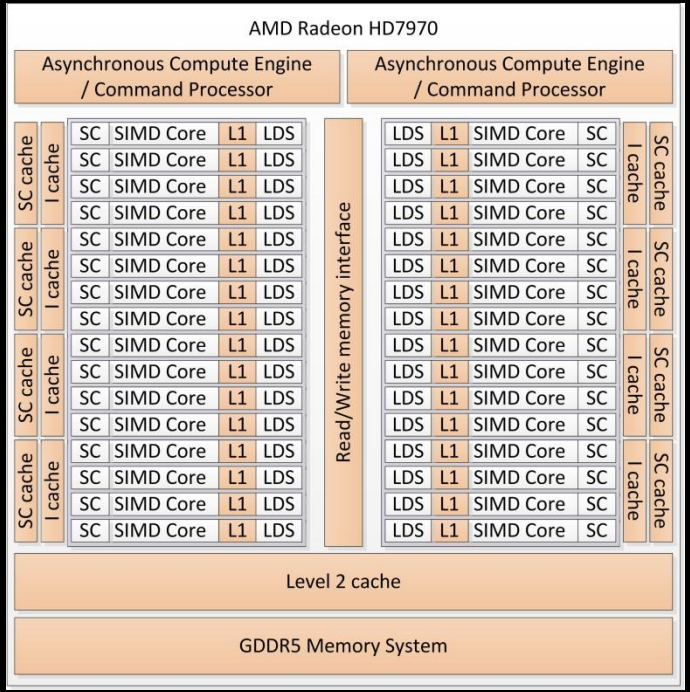
USE CASES FOR THIS ARE FAIRLY OBVIOUS

- Pointer chasing algorithms with mixed GPU/CPU use
- Algorithms that construct data on the CPU, use it on the GPU
- Allows for more fine-grained data use without explicit copies
- Covers cases where explicit copies are difficult:
  - Picture OS allocated data that the OpenCL runtime doesn't know about
- However, that wasn't quite enough to achieve our goals…

SO WHAT ELSE DO WE NEED?

SO WHAT ELSE DO WE NEED?

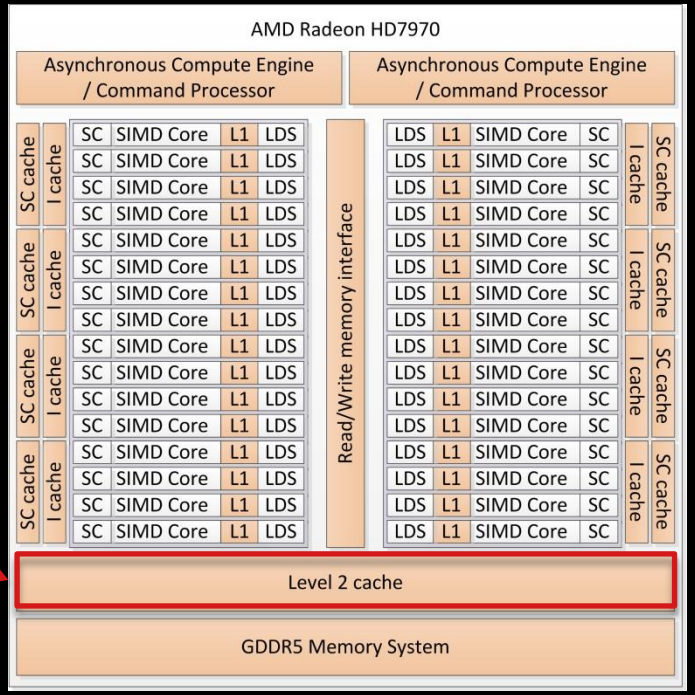
- We need a global view of the GPU, not just of the shader cores



SO WHAT ELSE DO WE NEED?

- We need a global view of the GPU, not just of the shader cores

Of course, we need to see the data here too

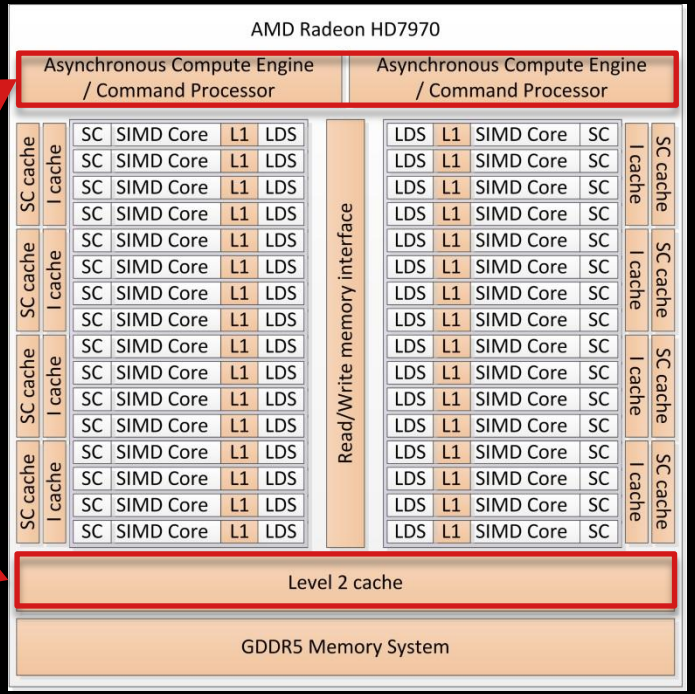


SO WHAT ELSE DO WE NEED?

- We need a global view of the GPU, not just of the shader cores

Most importantly! We need to be able to compute on the same data here.

Of course, we need to see the data here too

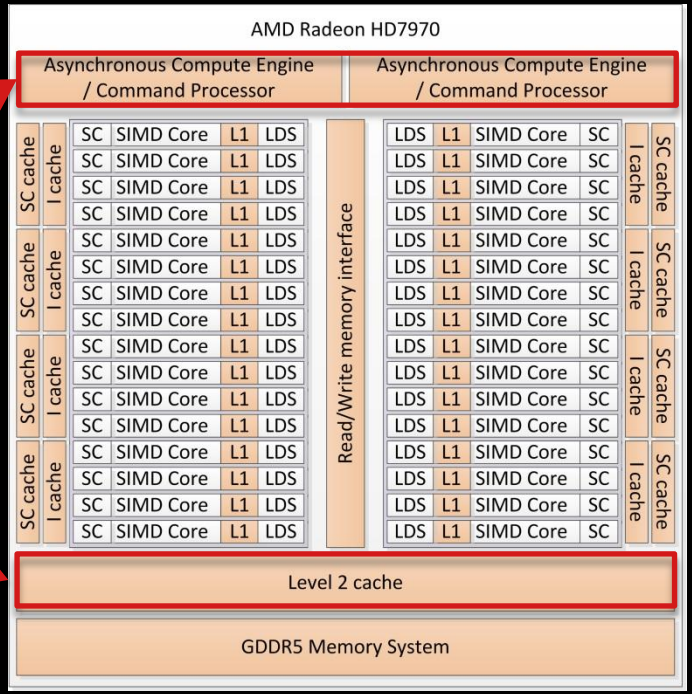


SO WHAT ELSE DO WE NEED?

- We need a global view of the GPU, not just of the shader cores

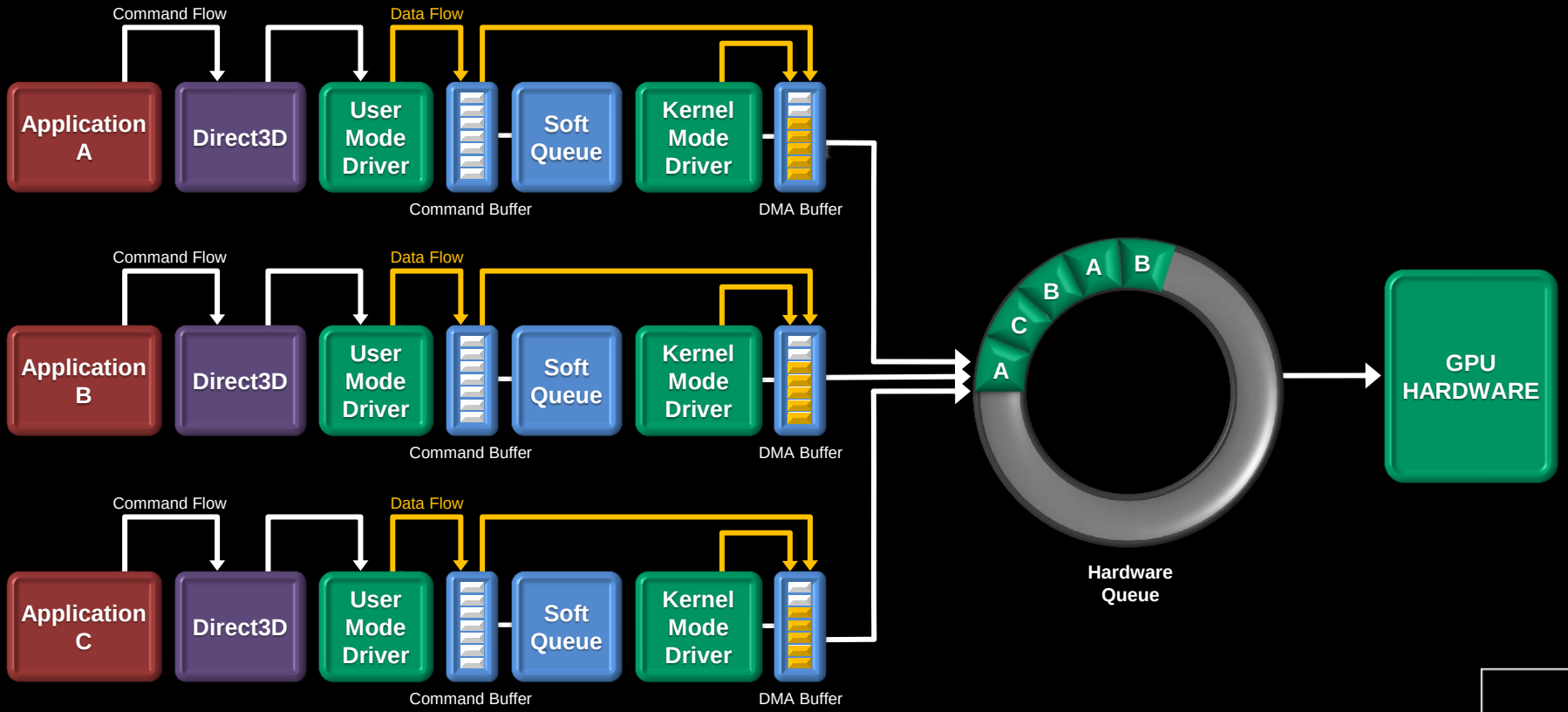
Most importantly! We need to be able to compute on the same data here.

Of course, we need to see the data here too

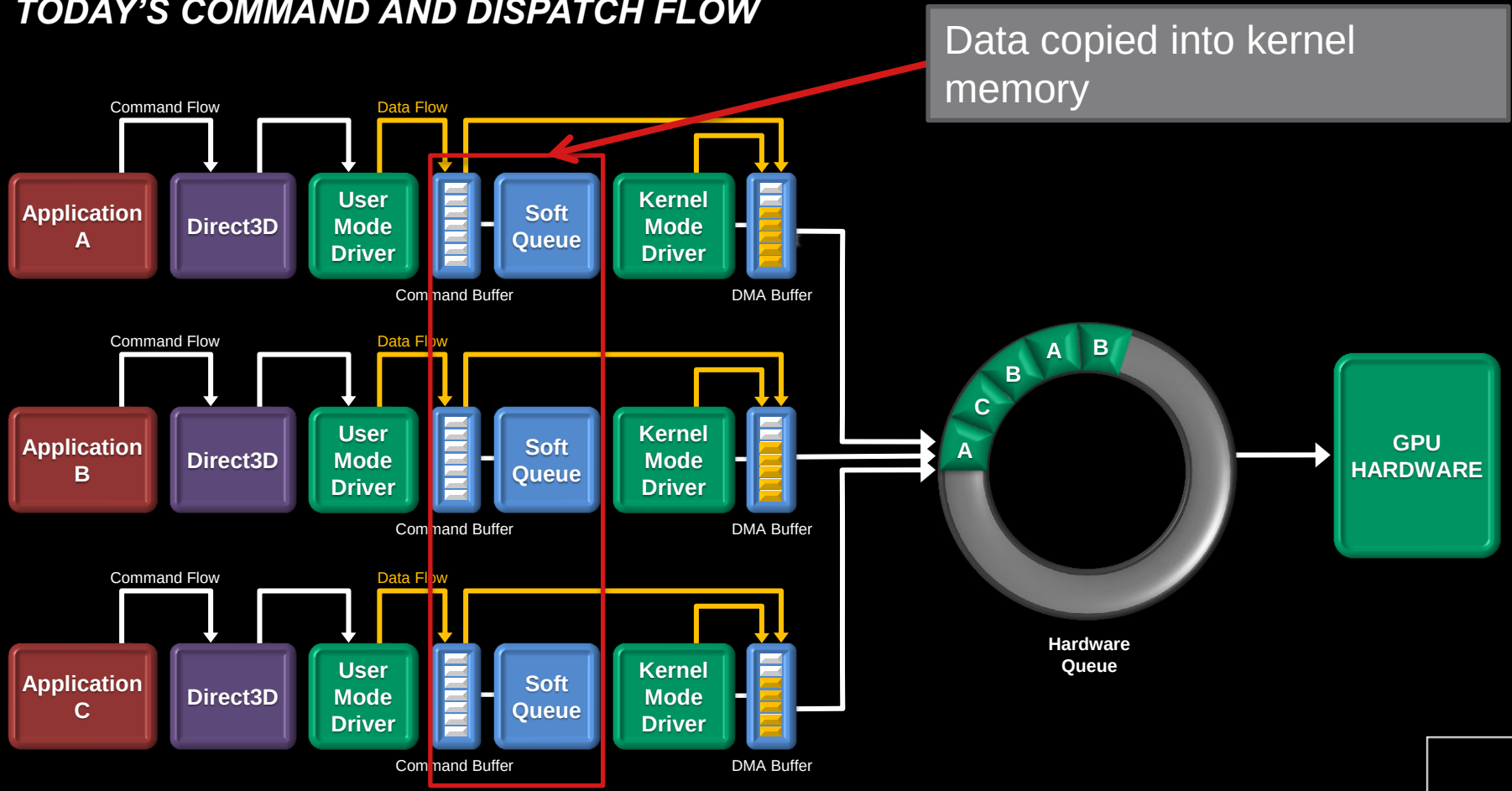


- Let's look at how GPU work dispatch works currently

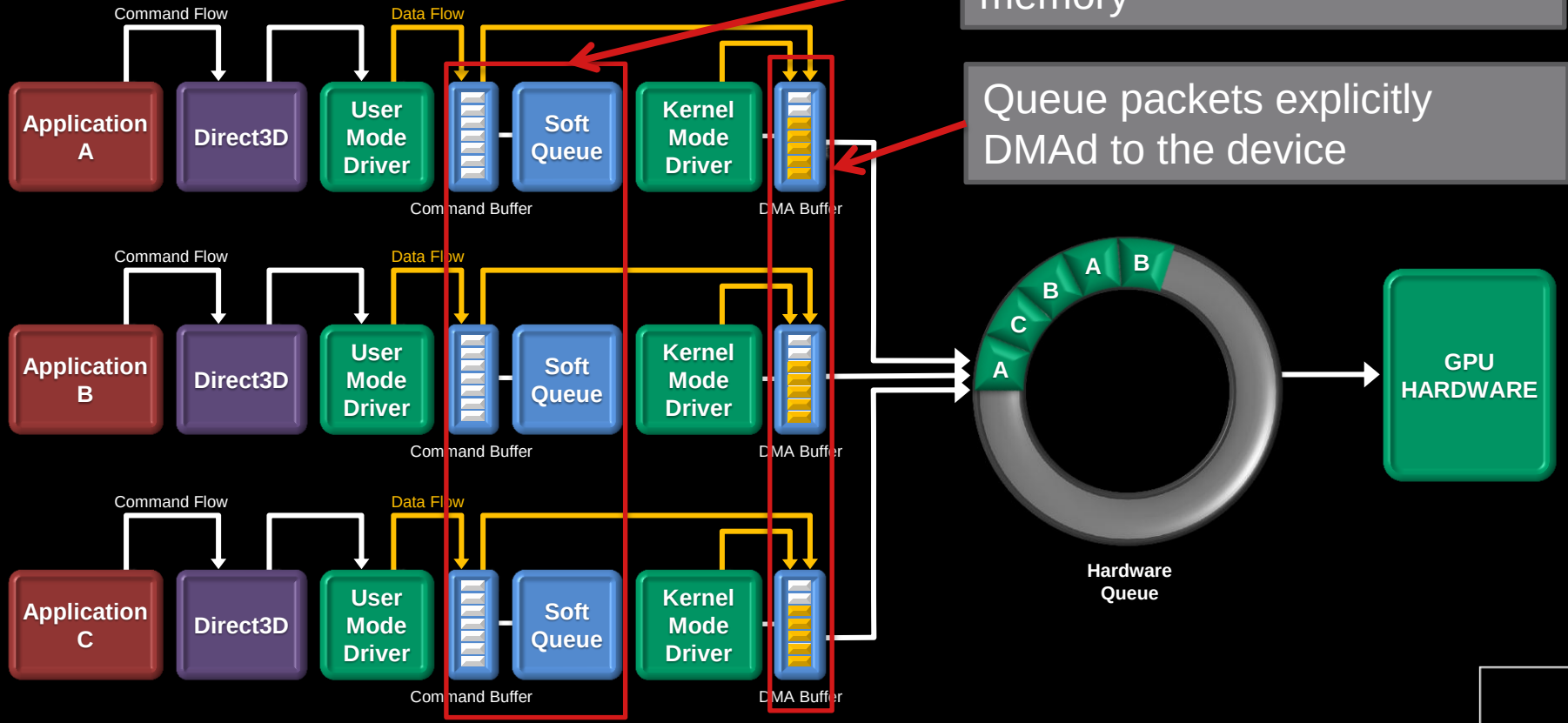
TODAY'S COMMAND AND DISPATCH FLOW



TODAY'S COMMAND AND DISPATCH FLOW

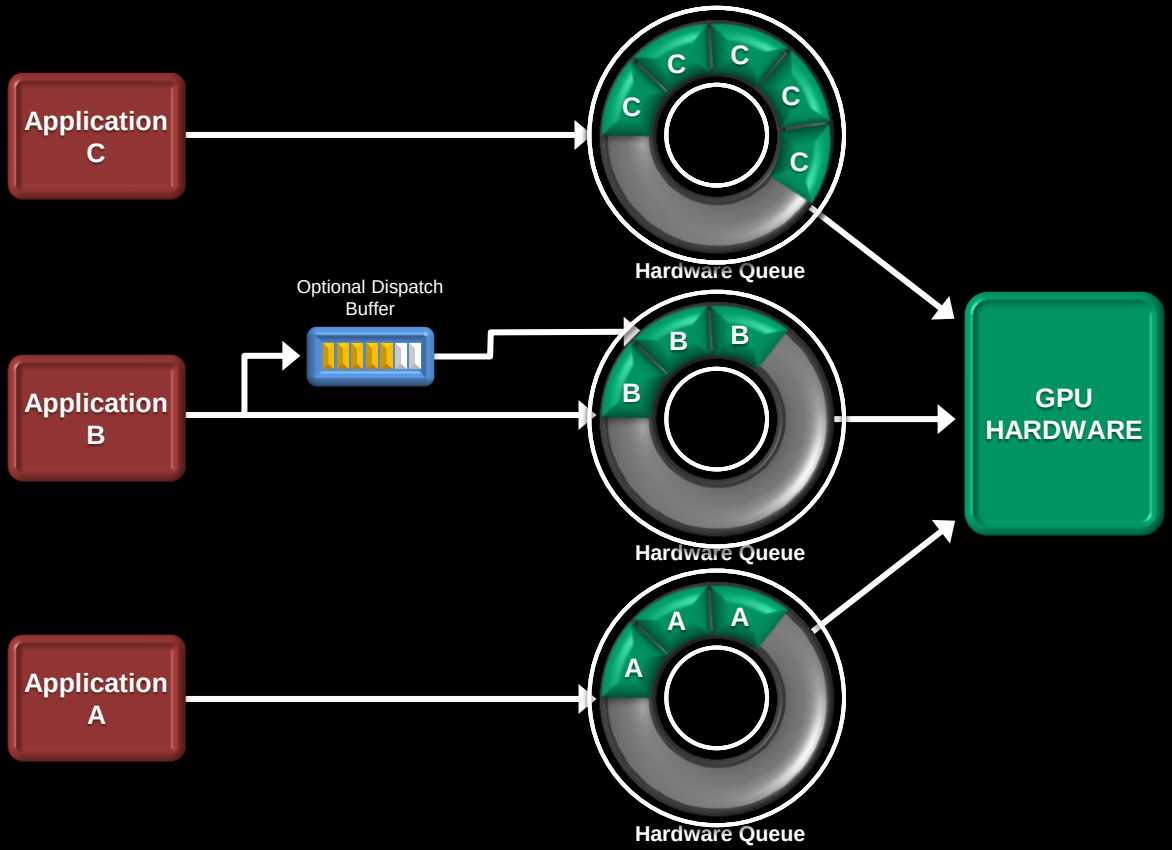


TODAY'S COMMAND AND DISPATCH FLOW

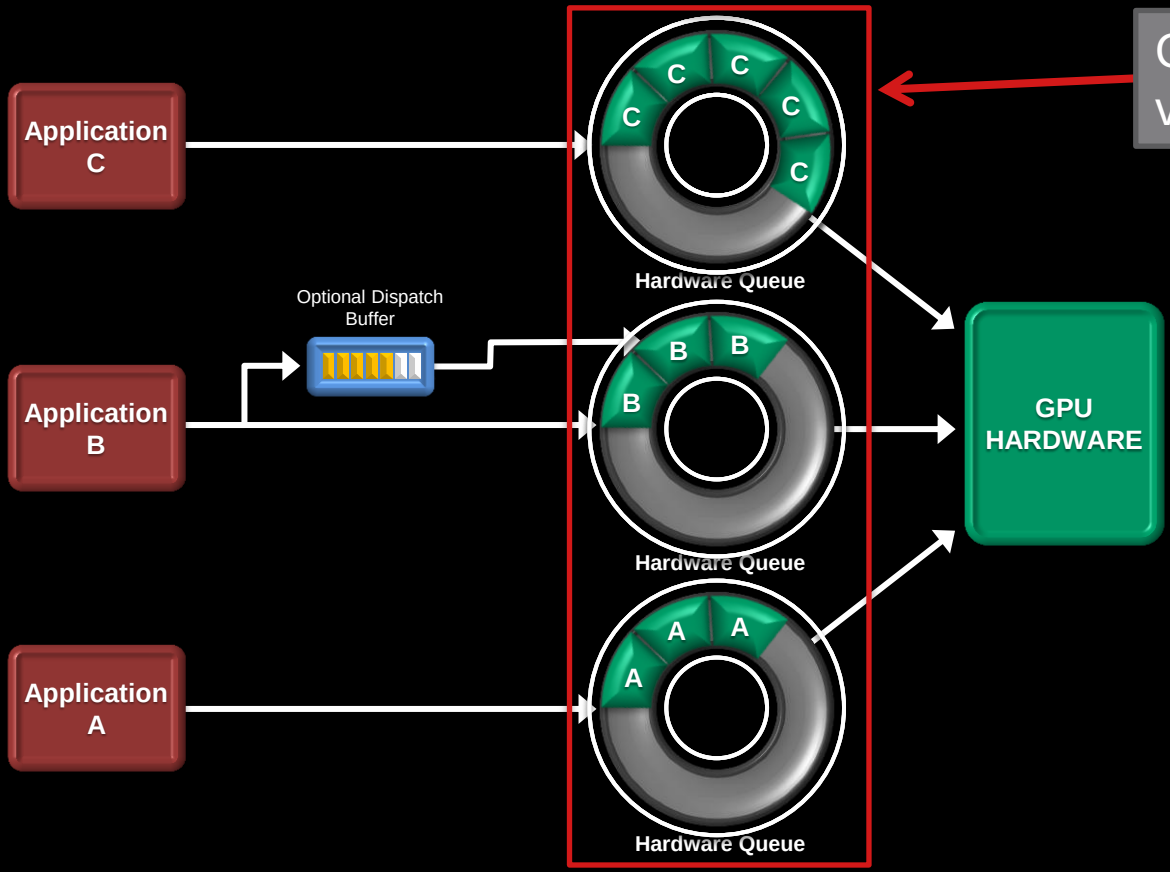


AMD

FUTURE COMMAND AND DISPATCH FLOW



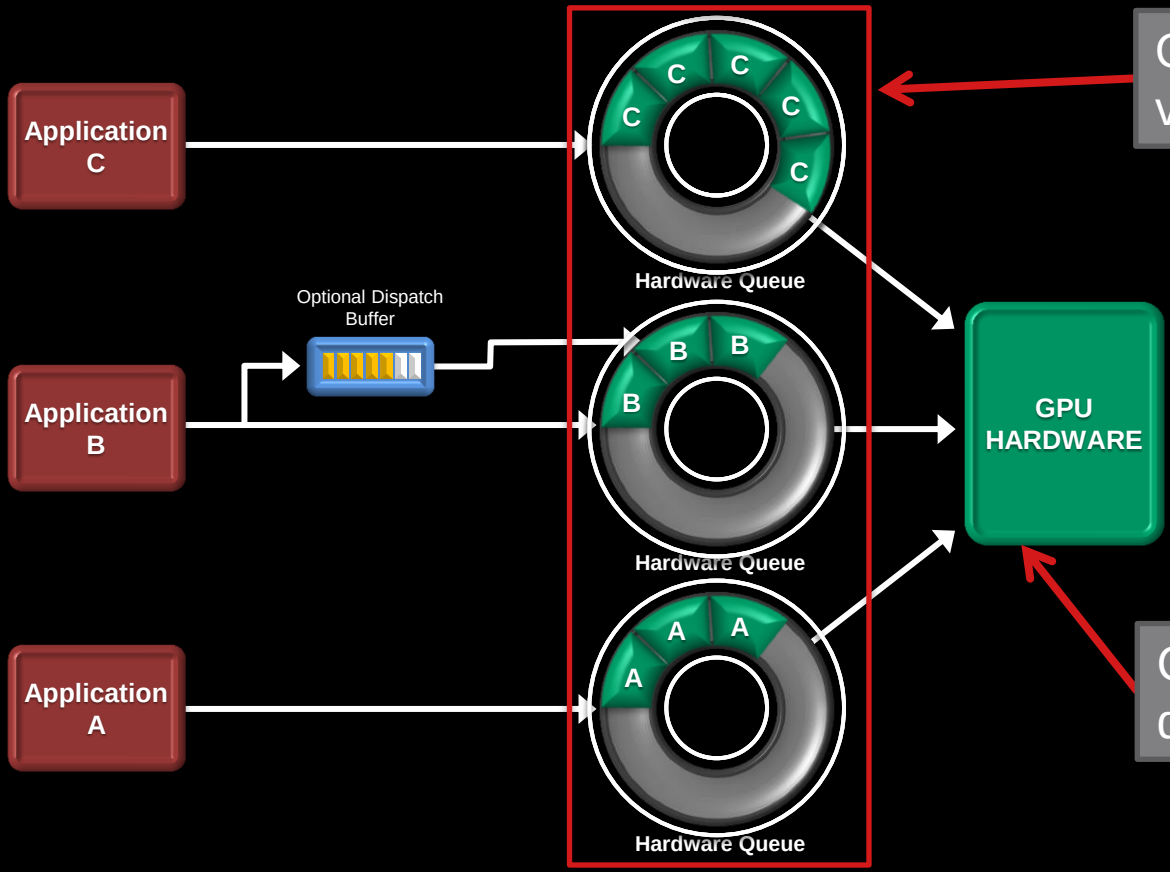
FUTURE COMMAND AND DISPATCH FLOW



Queues are in user-mode virtual memory

AMD

FUTURE COMMAND AND DISPATCH FLOW

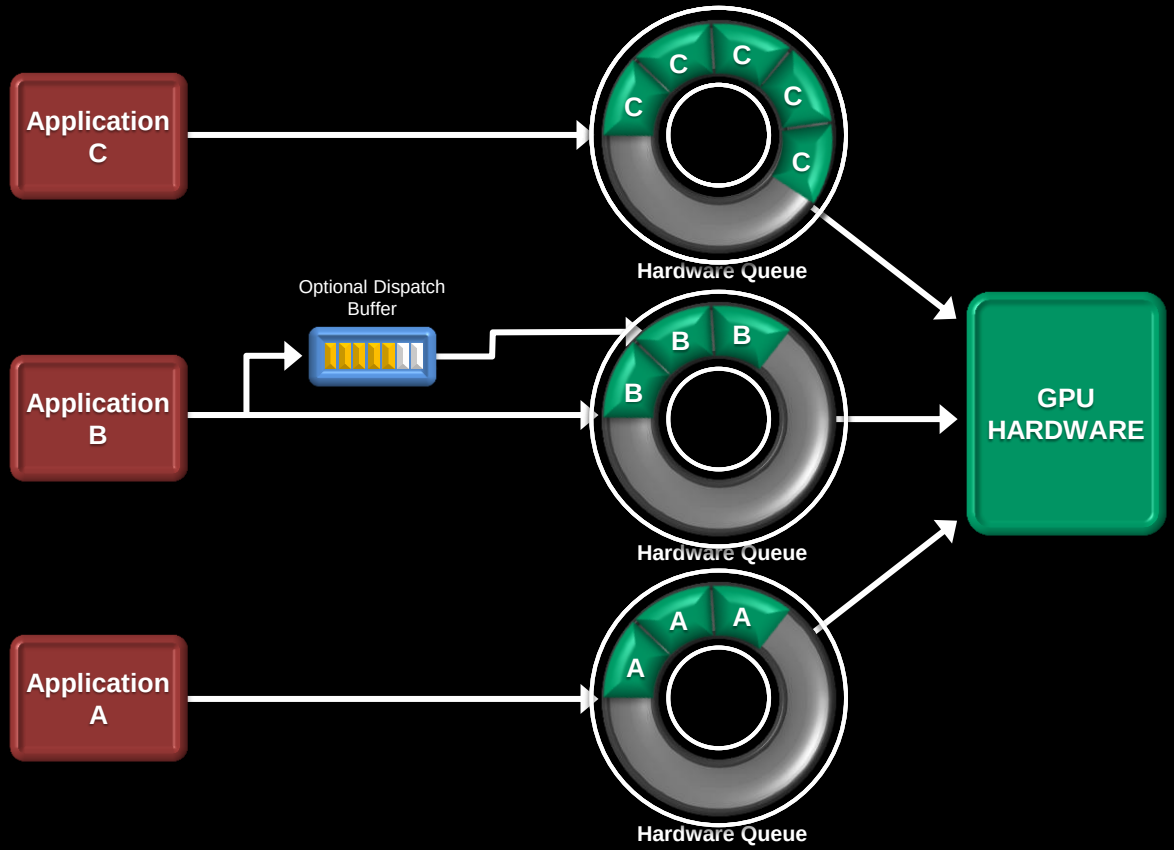


Queues are in user-mode virtual memory

Command processor can read queues directly

AMD

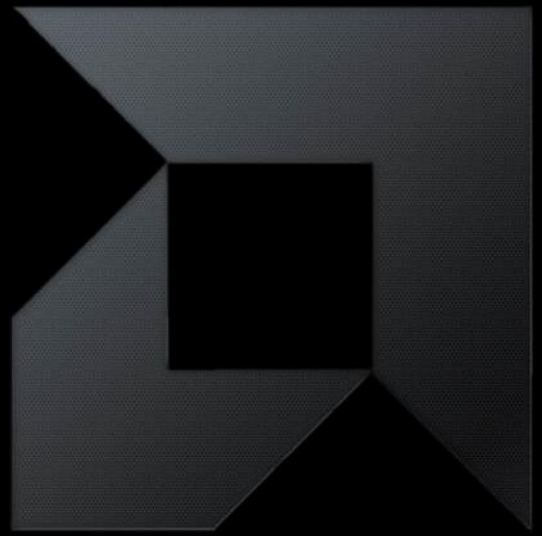
FUTURE COMMAND AND DISPATCH FLOW



- Application codes to the hardware
- User mode queuing
- Hardware scheduling
- Low dispatch times

- No required APIs
- No Soft Queues
- No User Mode Drivers
- No Kernel Mode Transitions
- No Overhead!

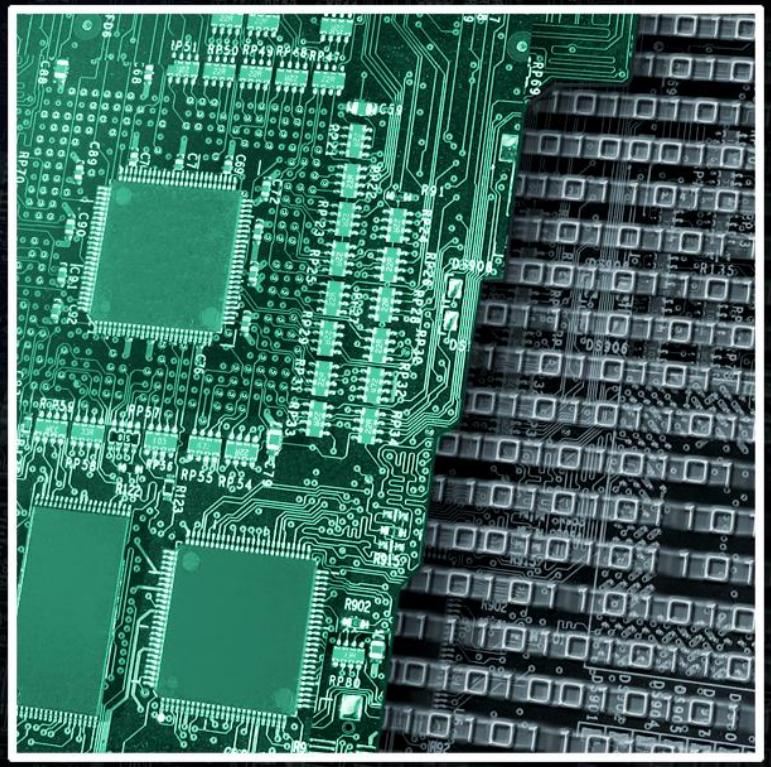
ARCHITECTED ACCESS



AMD

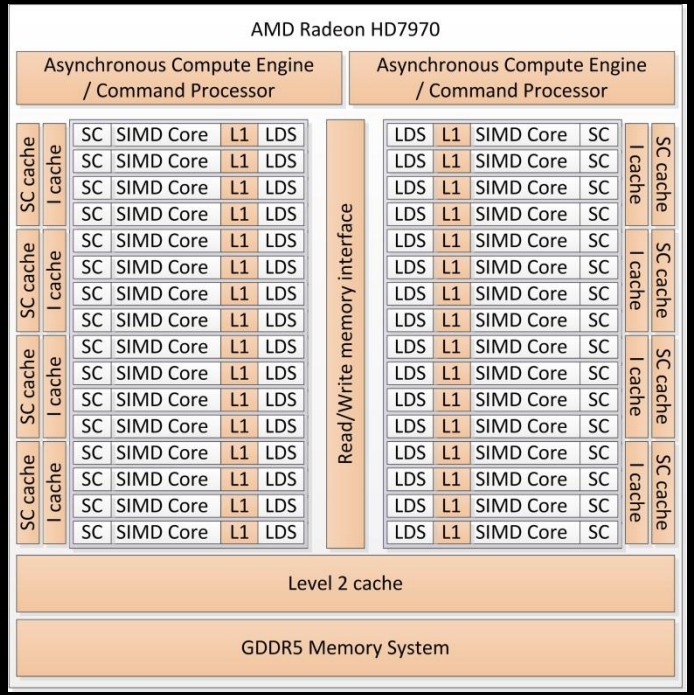
HETEROGENEOUS SYSTEM ARCHITECTURE – AN OPEN PLATFORM

- Open Architecture, published specifications
  - HSAIL virtual ISA
  - HSA memory model
  - Architected Queuing Language
- HSA system architecture
  - Inviting partners to join us, in all areas
  - Hardware companies
  - Operating Systems
  - Tools and Middleware
  - Applications
- HSA Foundation being formed



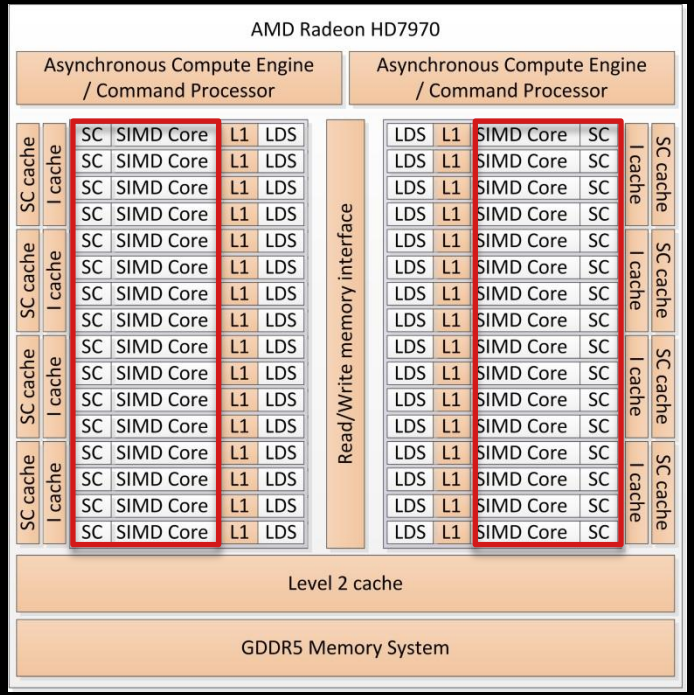
ARCHITECTED INTERFACES

- Standardize interfaces to features of the system
  - The compute cores
  - The memory hierarchy
  - Work dispatch
- Standardize access to the device
  - Memory backed queues
  - User space data structures



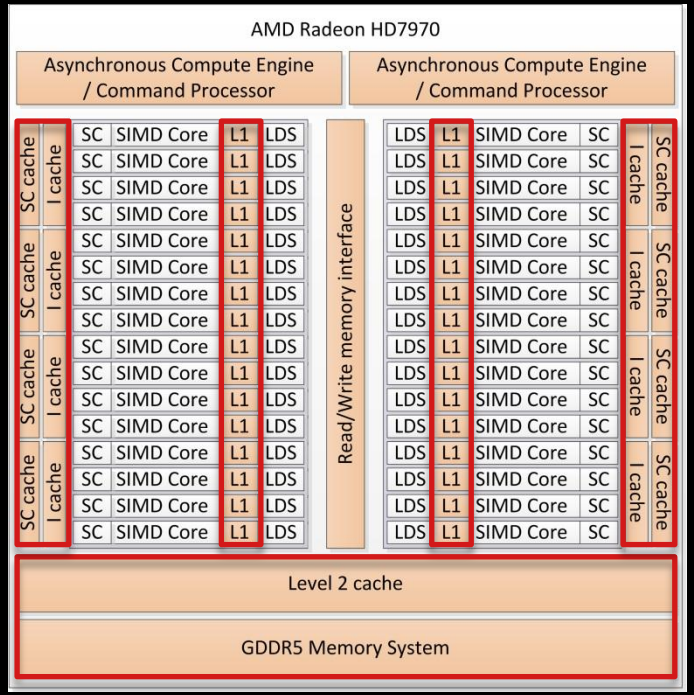
HSA INTERMEDIATE LAYER - HSAIL

- HSAIL is a virtual ISA for parallel programs
  - Finalized to ISA by a runtime compiler or "Finalizer"
- Explicitly parallel
  - Designed for data parallel programming
- Support for exceptions, virtual functions, and other high level language features
- Syscall methods
  - GPU code can call directly to system services, IO, printf, etc
- Debugging support



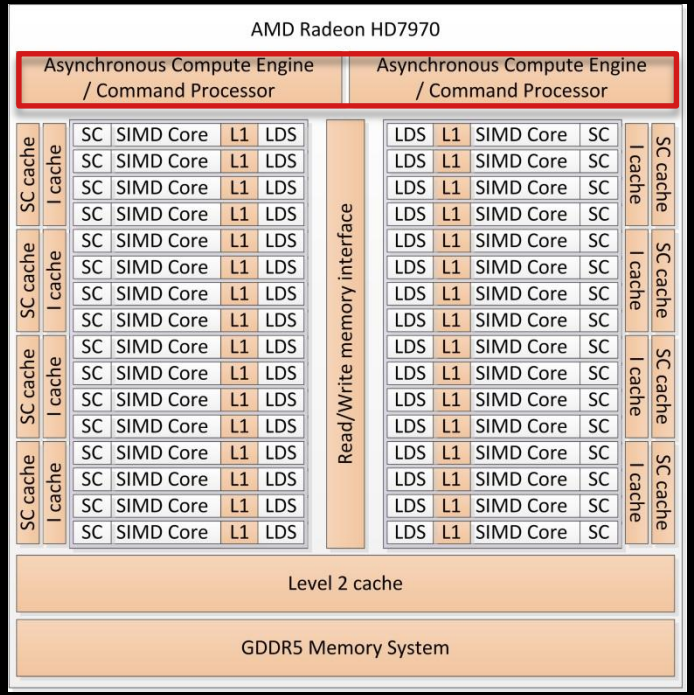
HSA MEMORY MODEL

- Designed to be compatible with C++11, Java and .NET Memory Models
- Relaxed consistency memory model for parallel compute performance
- Loads and stores can be re-ordered by the finalizer
- Visibility controlled by:
  - Load.Acquire
  - Store.Release
  - Barriers



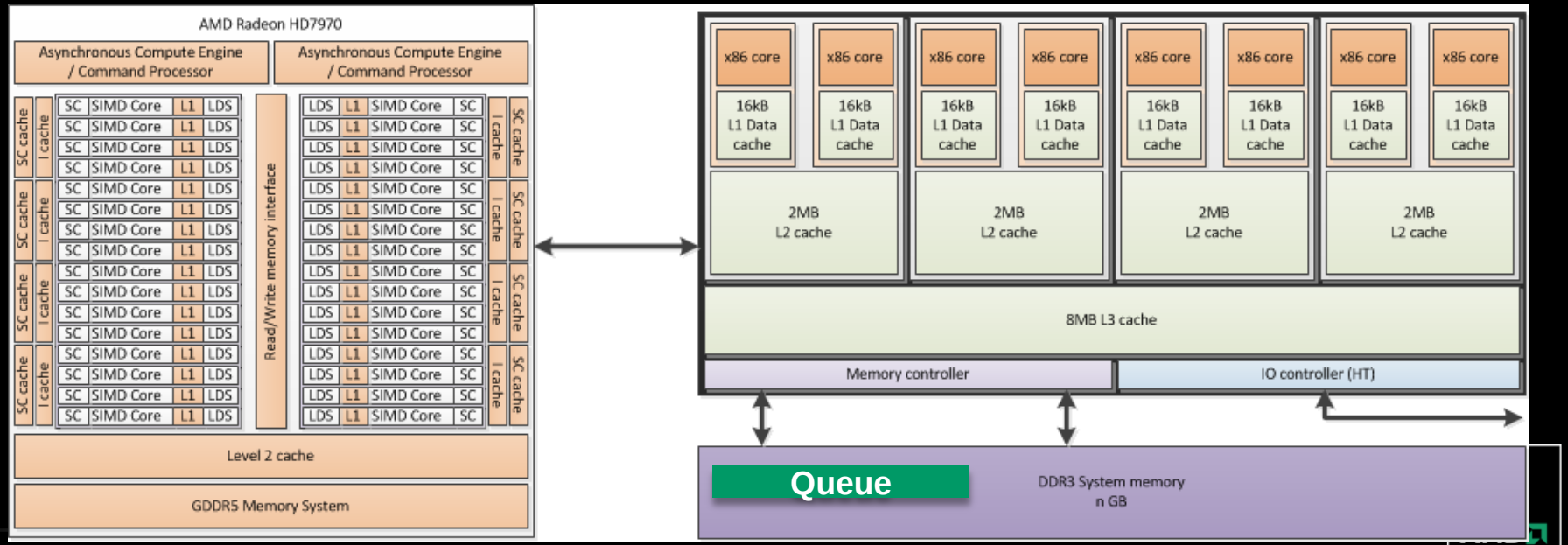
ARCHITECTED QUEUING LANGUAGE

- Defines dispatch characteristics in a small packet in memory
  - Platform neutral work offload
- Designed to be interpreted by the device
  - Firmware implementations
  - Or directly implemented in hardware



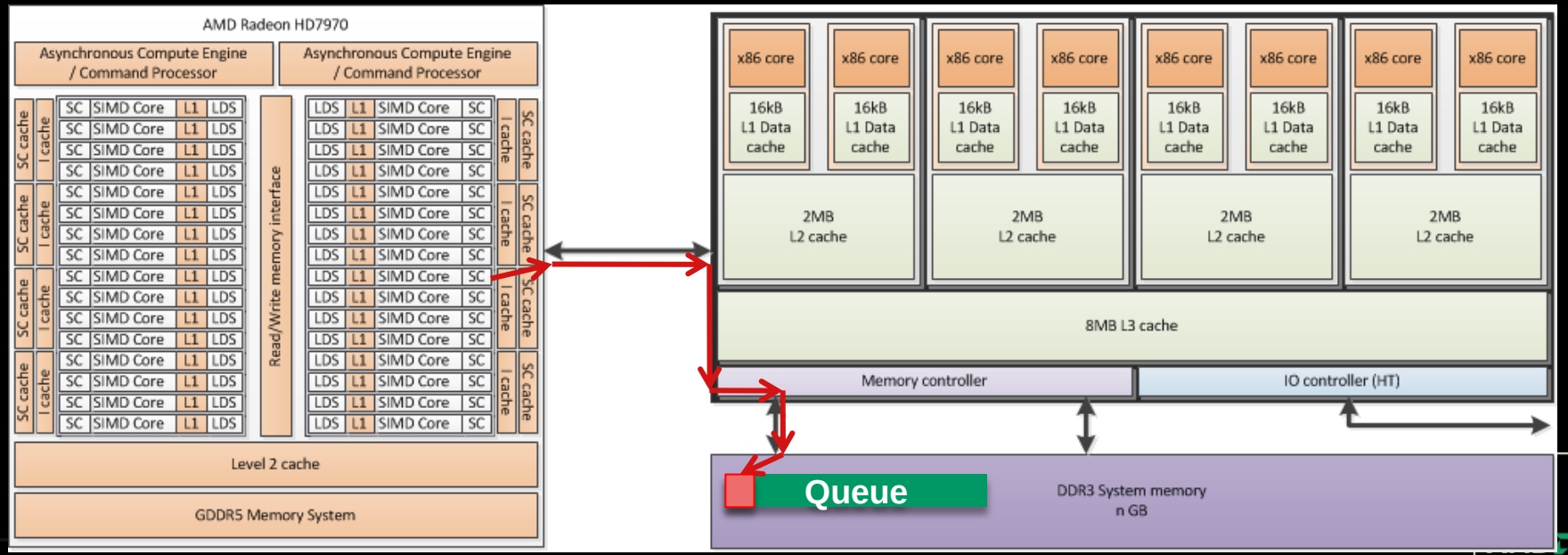
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



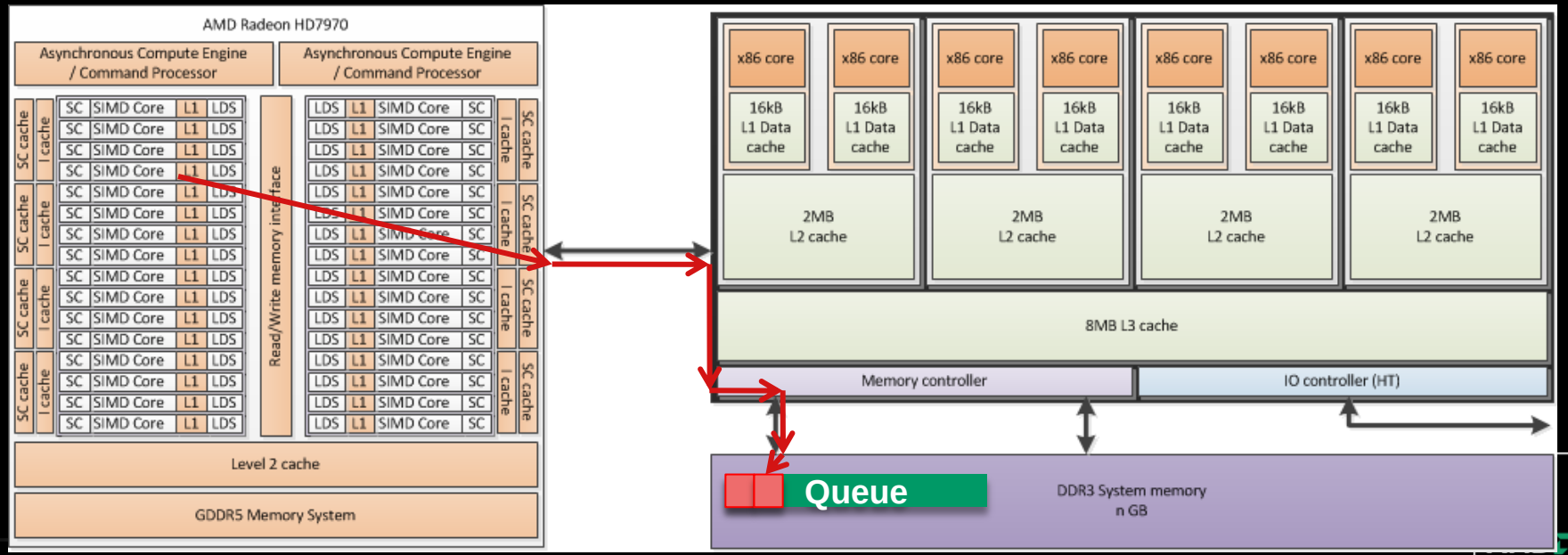
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



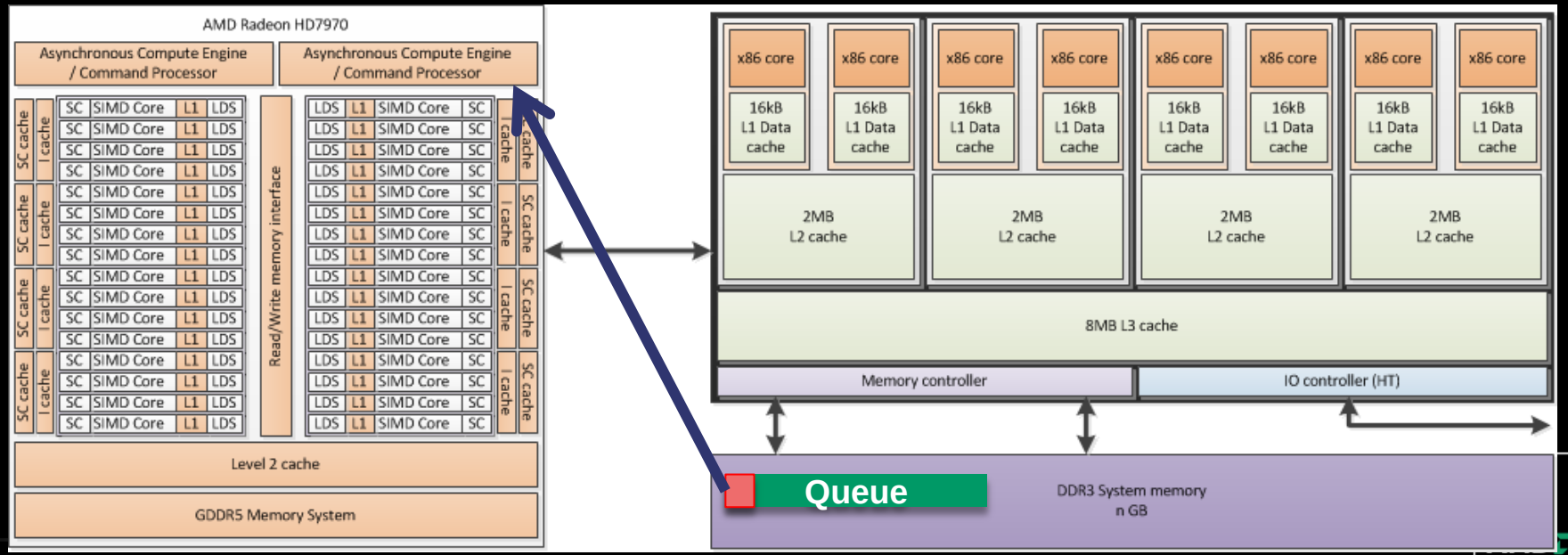
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



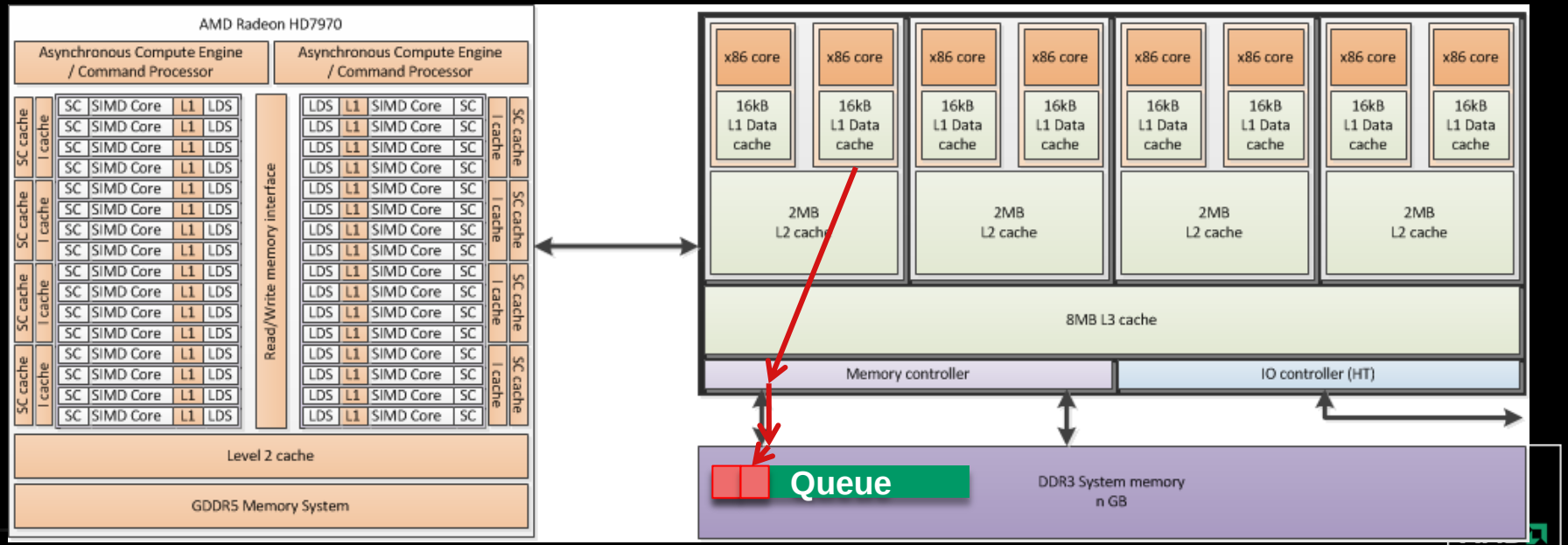
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



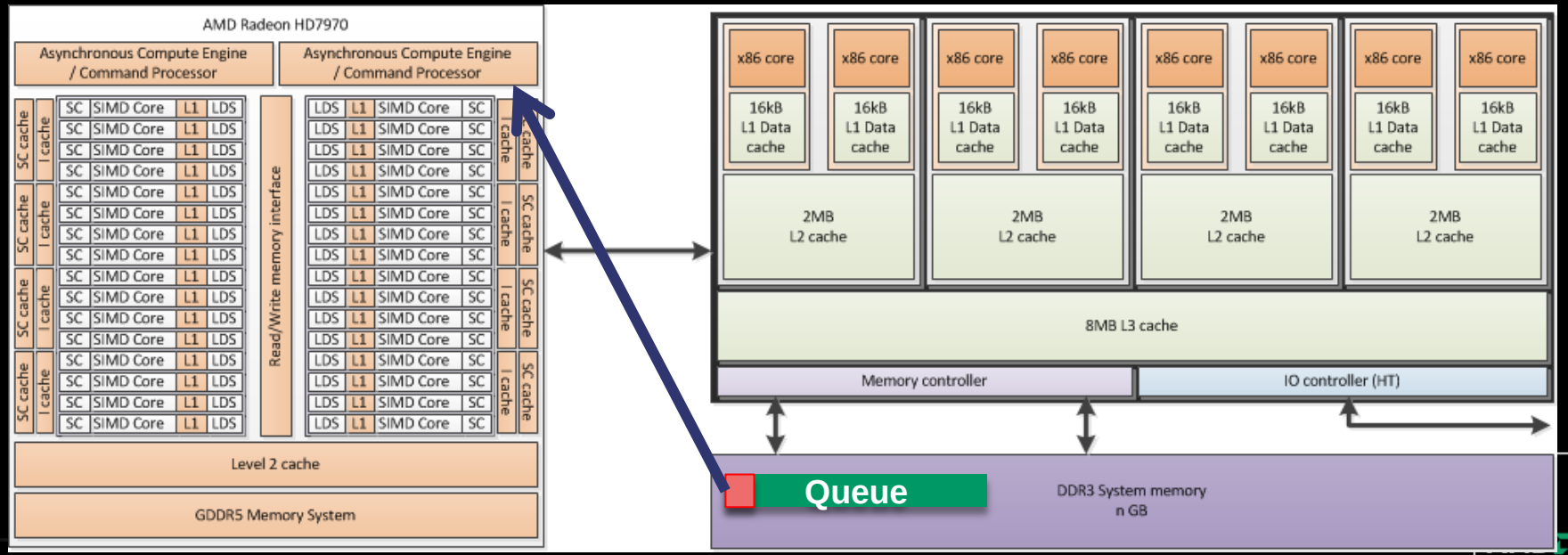
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



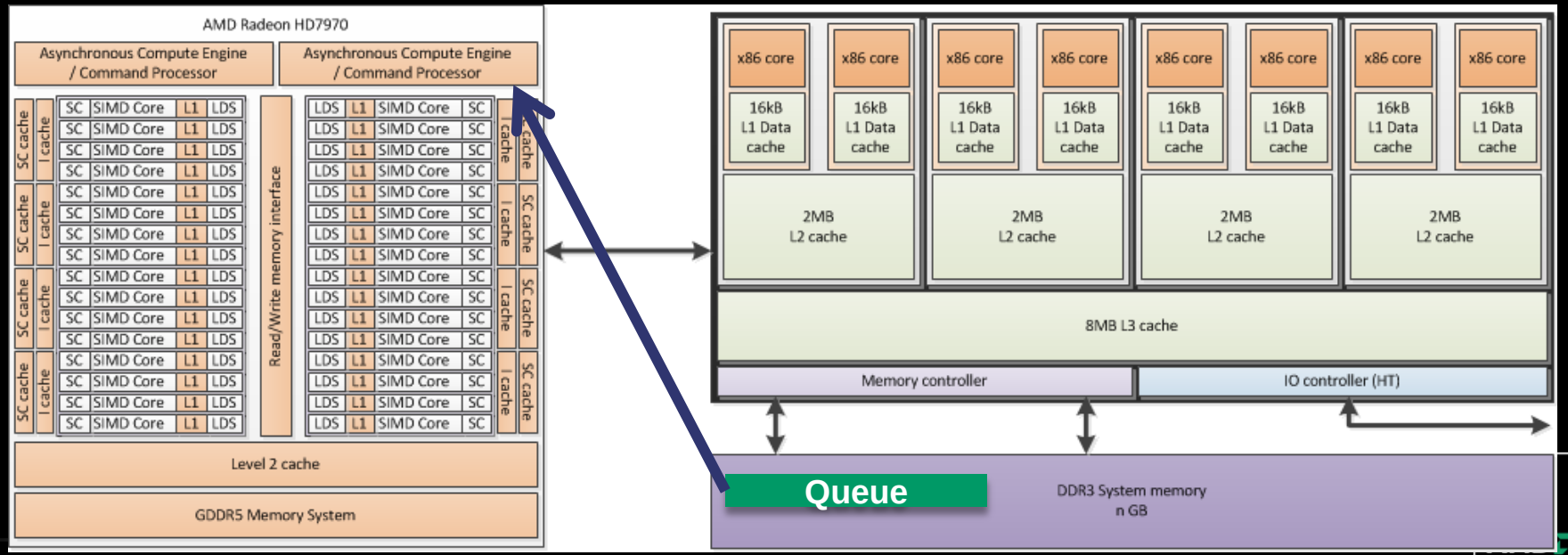
QUEUES

- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



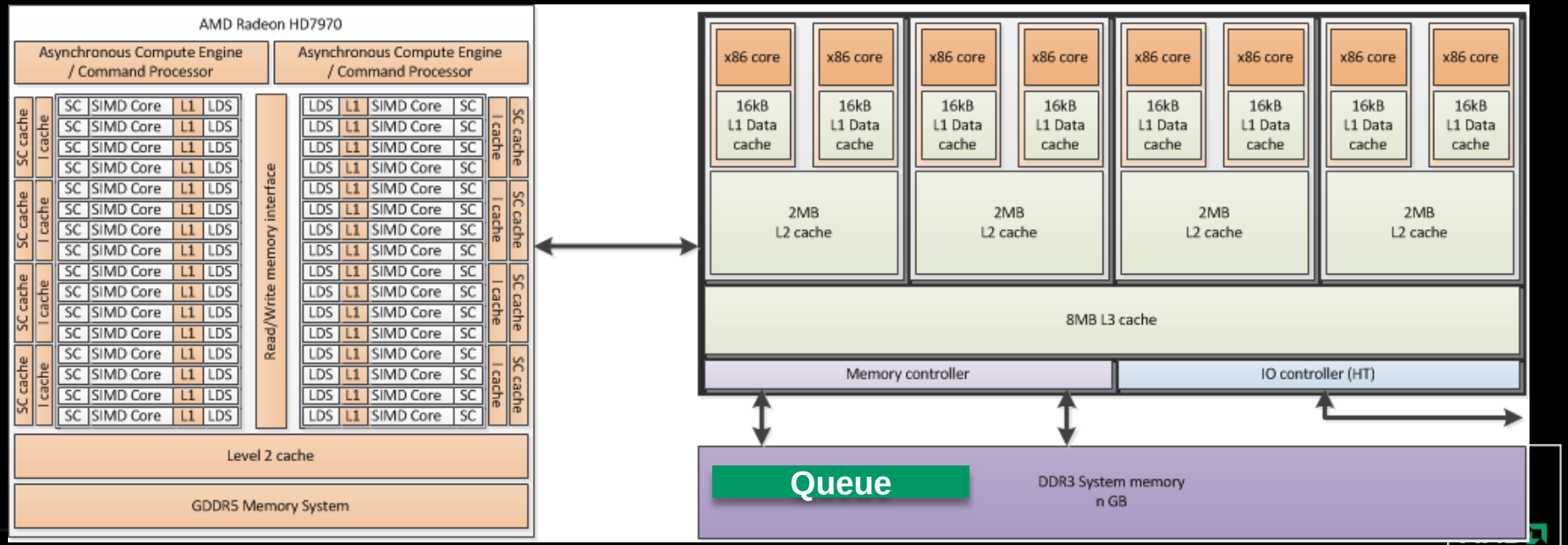
QUEUES

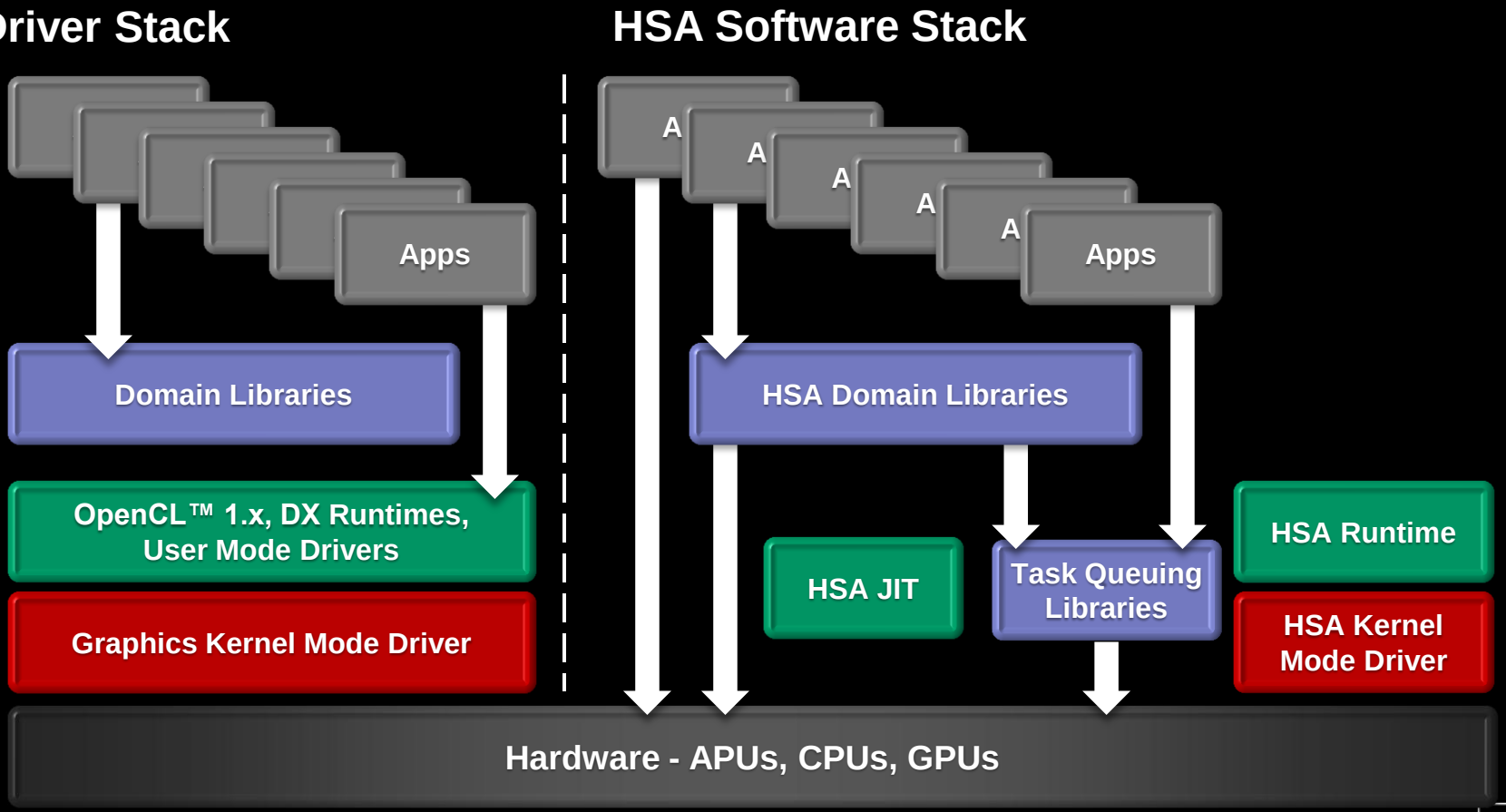
- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue



QUEUES

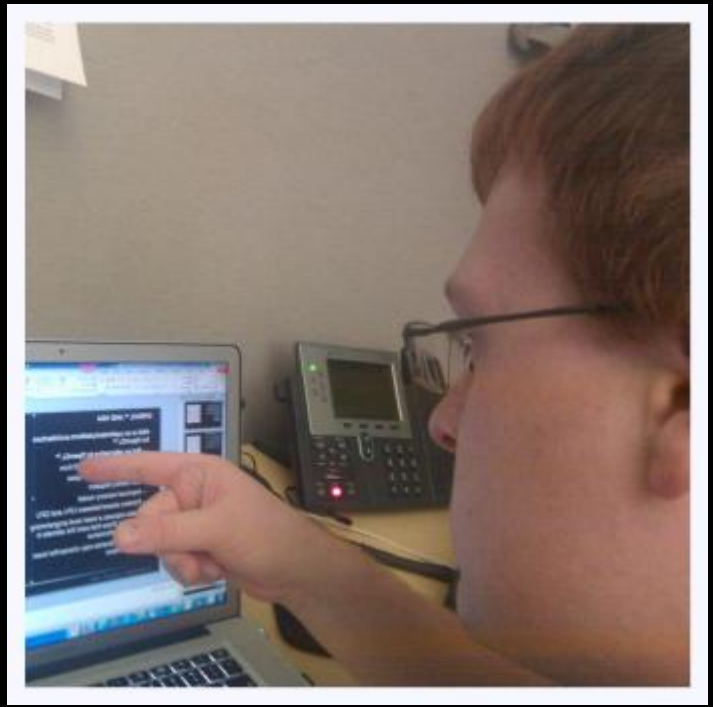
- User space memory allows queues to span devices
- Standardized packet format (AQL) enables flexible and portable use
- Single consumer, multiple producer of work
  - Enables support for task queuing runtimes and device->self enqueue




Driver Stack
HSA Software Stack
Apps
A
A
A
A
A
Apps
Domain Libraries
HSA Domain Libraries
OpenCL™ 1.x, DX Runtimes,
User Mode Drivers
HSA JIT
Task Queuing
Libraries
HSA Runtime
Graphics Kernel Mode Driver
HSA Kernel
Mode Driver
Hardware - APUs, CPUs, GPUs
AMD user mode component
AMD kernel mode component
All others contributed by third parties or AMD

OPENCL™ AND HSA

- **HSA is an optimized platform architecture for OpenCL™**
  - **Not an alternative to OpenCL™**
- OpenCL™ on HSA will benefit from
  - Avoidance of wasteful copies
  - Low latency dispatch
  - Improved memory model
  - Pointers shared between CPU and GPU
- HSA also exposes a lower level programming interface, for those that want the ultimate in control and performance
  - Optimized libraries may choose the lower level interface



QUESTIONS

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.