HSA and the modern GPU

LEE HOWES
JUNE 03, 2013

HSA AND THE MODERN GPU

AMD

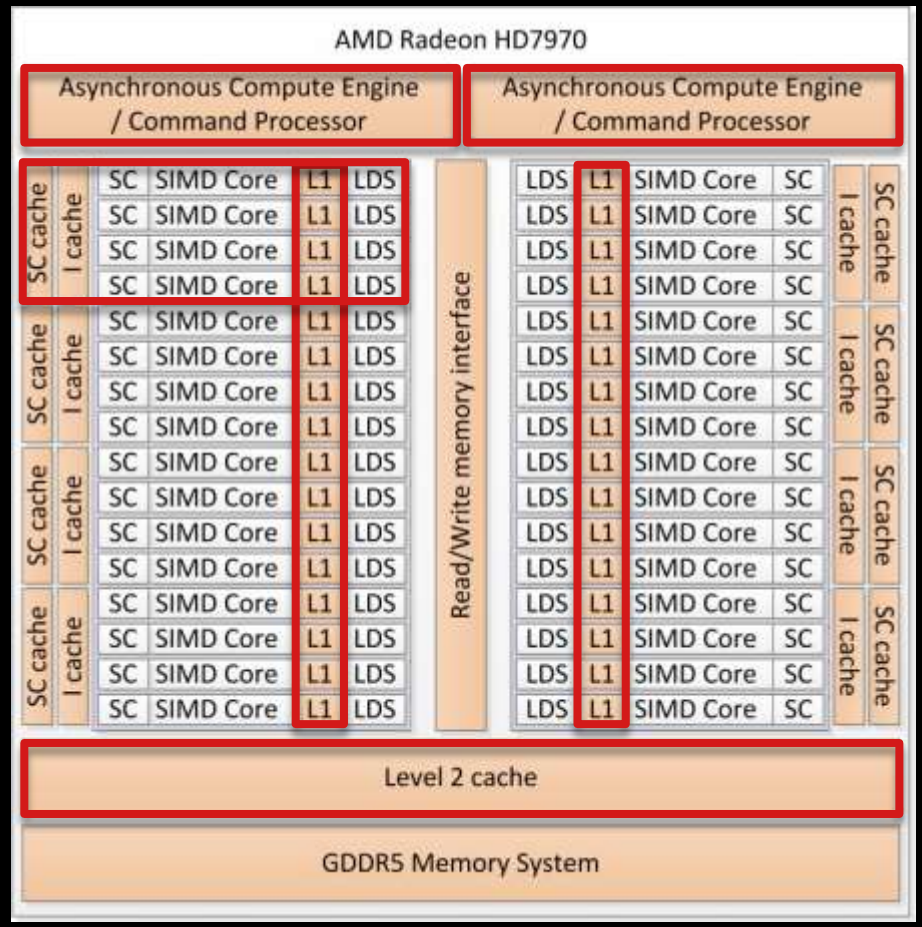
- In this brief talk we will cover three topics:
  - Changes to the shader core and memory system
  - Changes to the use of pointers
  - Architected definitions to use these new features

- We'll look both at how the hardware is becoming more flexible and give some idea about why

The HD7970 and Graphics Core Next

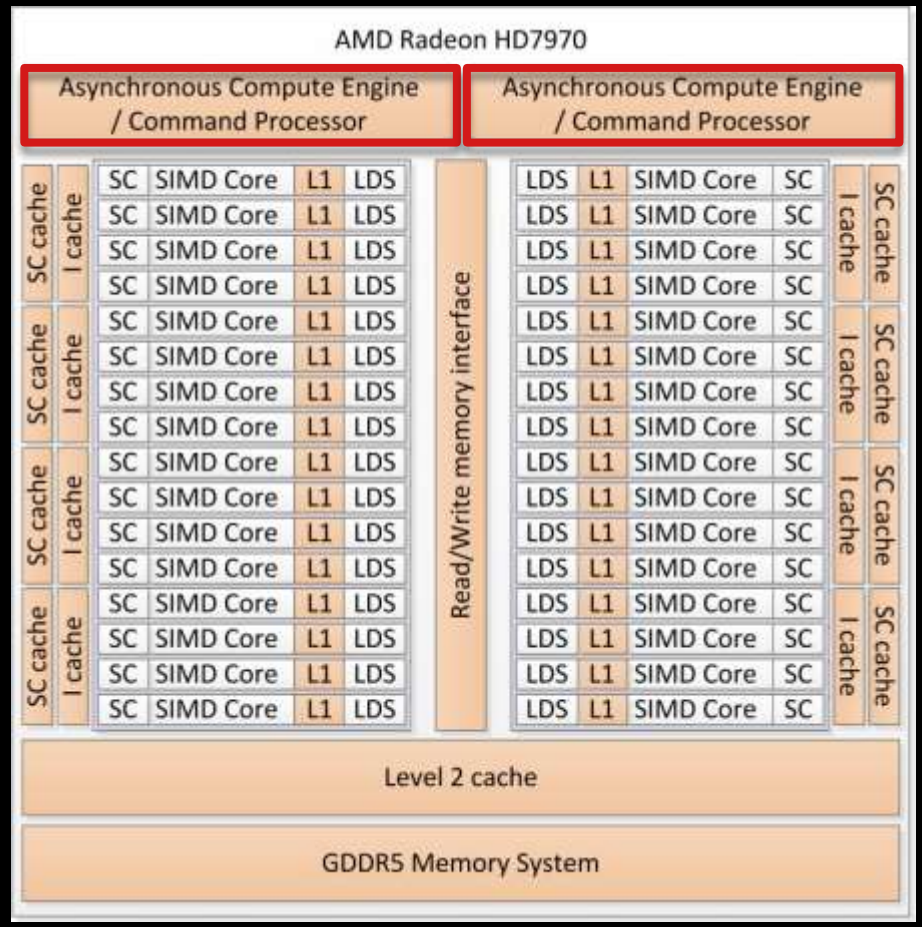
AMD RADEON™ HD7970 - GLOBALLY

- A multi-core superscalar parallel processing engine
- Two command processors
  - Capable of processing two command queues concurrently
- Full read/write cache hierarchy
- 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

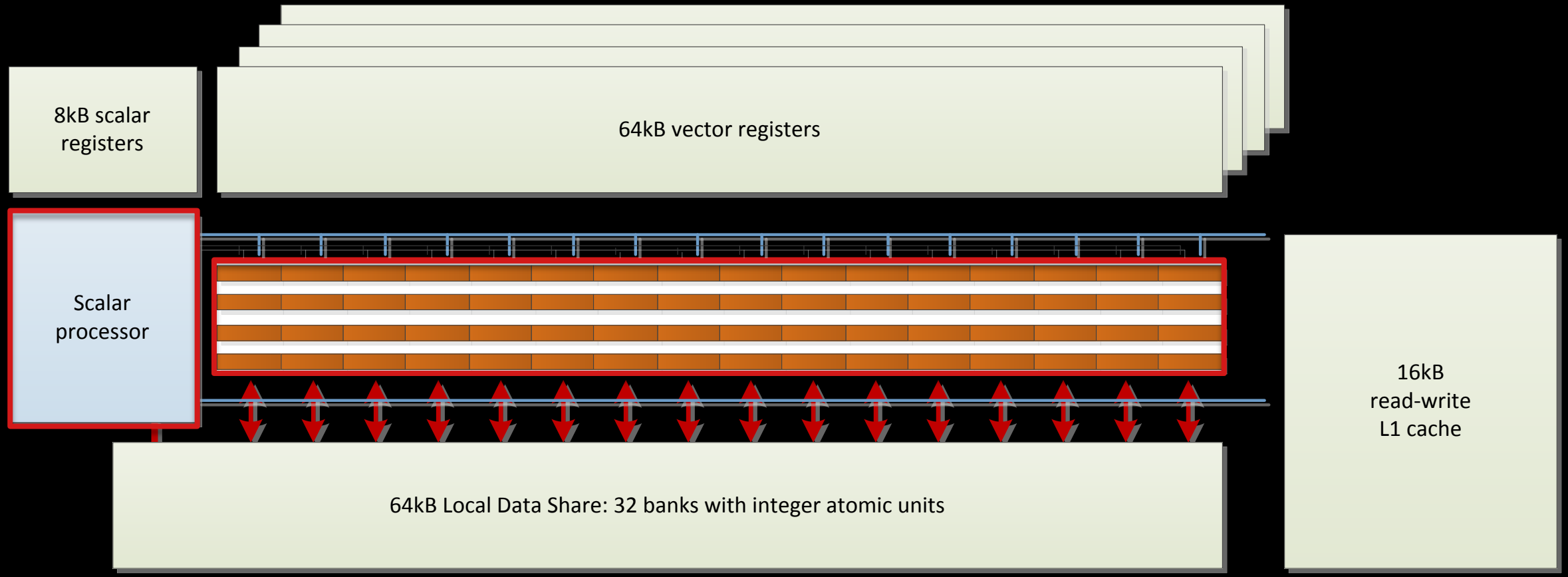


COMMAND PROCESSORS

- Consume input packets
  - Compute packets for most of us, (currently) generated from OpenCL™ commands
  - Graphics packets follow a similar path
- Multiple queues processed
  - So multiple queues can be in progress simultaneously
  - Enables concurrent execution of kernels
- Command processor instructs a scheduler to generate work
  - Scheduler generates work to place on the machine as capacity is available
  - Work is generated in the form of wavefronts
  - A wavefront is analogous to a CPU thread and represents 64 OpenCL work items, or 64 parallel instances of the OpenCL kernel program as written.



THE SIMD CORE



- The heart of Graphics Core Next:
  - A scalar processor with four 16-wide vector units
  - Each lane of the vector unit is a full single precision floating point unit
- Caches and a very large register file

MAPPING FROM THE PROGRAMMING LEVEL

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

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

- Which we flatten via a sequence of compilation steps to a GPU ISA:

```
    v_cmp_gt_f32        r0,r1
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
label0:
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
label1:
    s_mov_b64           exec,s0
```

SIMD EXECUTION

- You are probably aware that in reality, work items aren't threads.

- The majority of modern 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 mask management to handle divergent control flow across the vector

- So trivially, you might imagine something like the following:

```
    v_cmp_gt_f32        r0,r1
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
label0:
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
label1:
    s_mov_b64           exec,s0
```

```
    v_cmp_gt_f32        r0,r1
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
label0:
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
label1:
    s_mov_b64           exec,s0
```

```
    v_cmp_gt_f32        r0,r1
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
label0:
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
label1:
    s_mov_b64           exec,s0
```

ACTUALLY, A LITTLE MORE INTERESTING THAN THAT

- Look closely at that code:

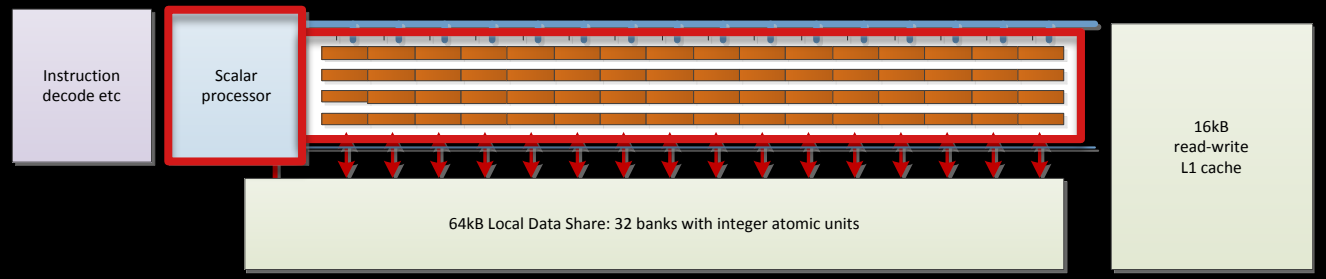
```
    v_cmp_gt_f32        r0,r1
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
label0:
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
label1:
    s_mov_b64           exec,s0
```

Scalar instructions

```
    s_mov_b64           s0,exec
    s_and_b64           exec,vcc,exec
    s_cbranch_vccz      label0
    s_andn2_b64         exec,s0,exec
    s_cbranch_execz     label1
    s_mov_b64           exec,s0
```

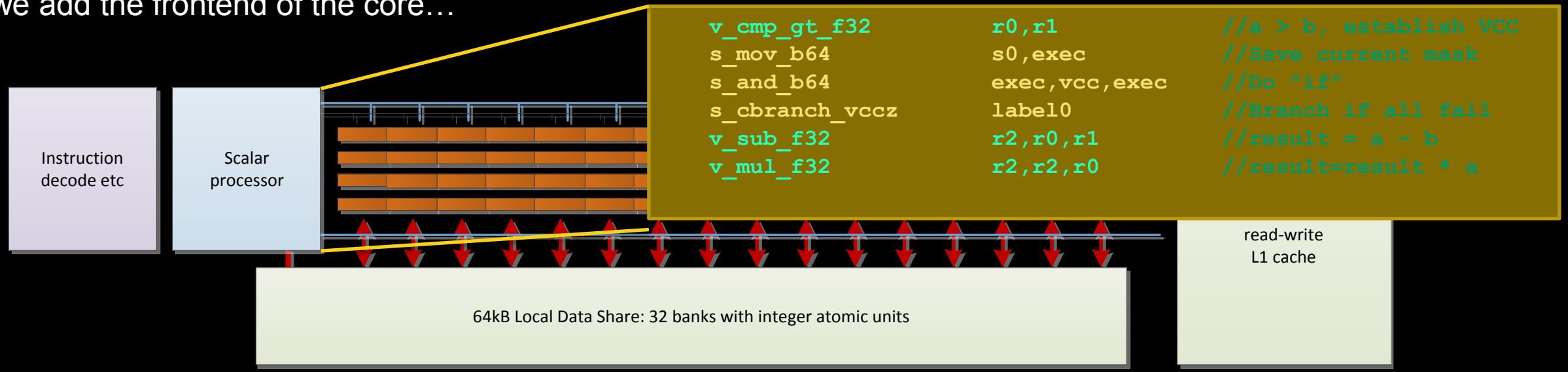
Vector instructions

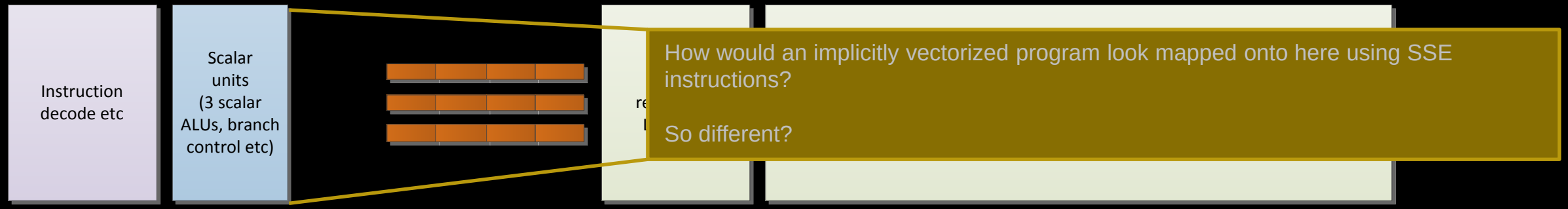
```
    v_cmp_gt_f32        r0,r1
    v_sub_f32           r2,r0,r1
    v_mul_f32           r2,r2,r0
    v_sub_f32           r2,r1,r0
    v_mul_f32           r2,r2,r1
```



FAMILIAR?

- If we add the frontend of the core…

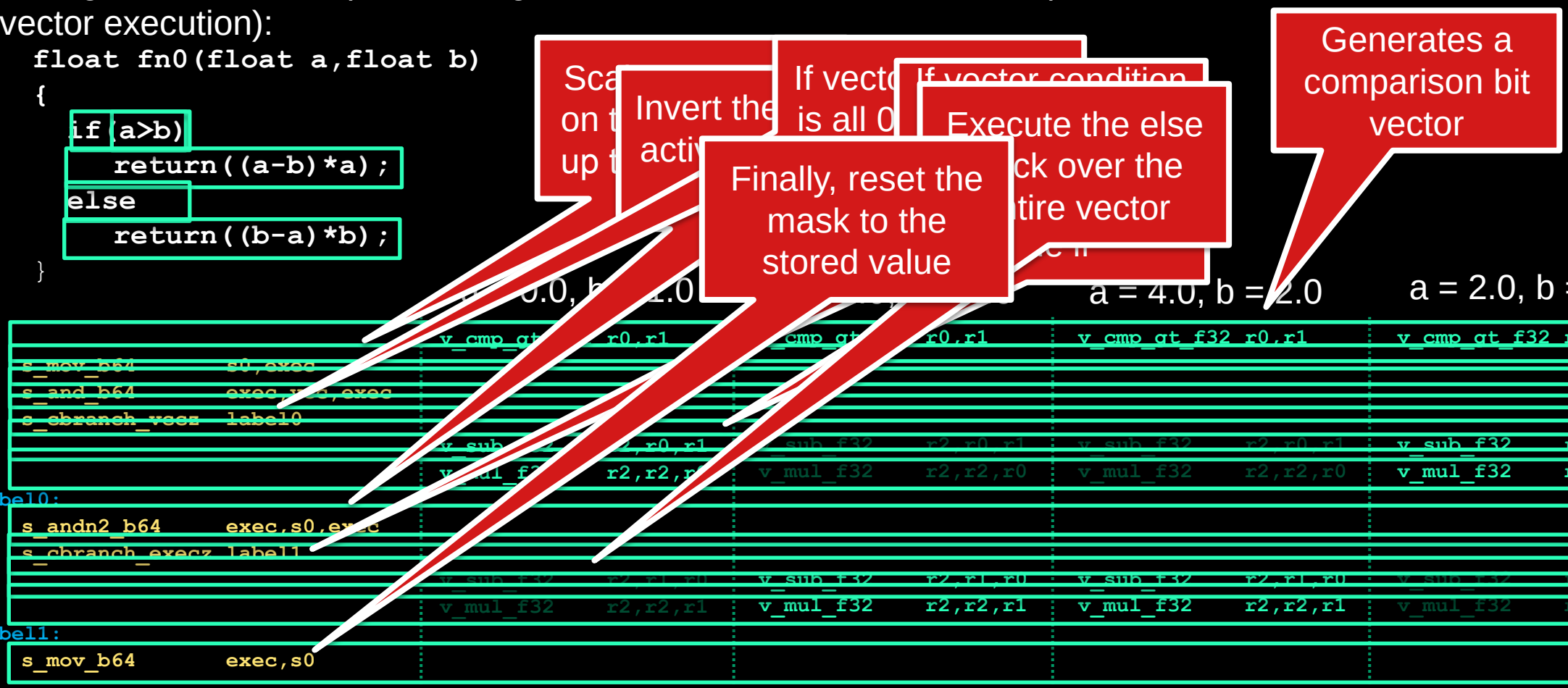




EXECUTION OVER A VECTOR

- Taking the same example, running over a fictional 4-element vector (the HD7970 uses a 64-element vector execution):

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



SOME LESSONS FROM THIS

- We could compile SPMD code in a similar fashion to a modern CPU:
  - In this case using SSE intrinsics
  - Mask operations performed directly on vector registers, but the effect is the same

```
   v_cmp_gt_f32       r0,r1
   s_mov_b64          s0,exec
   s_and_b64          exec,vcc,exec
   s_cbranch_vccz     label0

   v_sub_f32          r2,r0,r1
   v_mul_f32          r2,r2,r0
label0:
   s_andn2_b64        exec,s0,exec
   s_cbranch_execz    label1

   v_sub_f32          r2,r1,r0
   v_mul_f32          r2,r2,r1
label1:
   s_mov_b64          exec,s0
```

```
   vcc = _mm_cmpeq_epi32(r0, r1);
   exec = s0;
   exec = _mm_and_ps(vcc, exec);
   int a = _mm_movemask_ps(vcc);
   if( a ) goto label0
   r2 = _mm_sub_ps(r0, r1);
   r0 = _mm_mul_ps(r2, r2);
label0:
   exec = _mm_and_ps(s0, exec);
   int a = _mm_movemask_ps(exec);
   if( a ) goto label1
   r0 = _mm_sub_ps(r1, r2);
   r1 = _mm_mul_ps(r2, r2);
label1:
   s0 = exec;
```

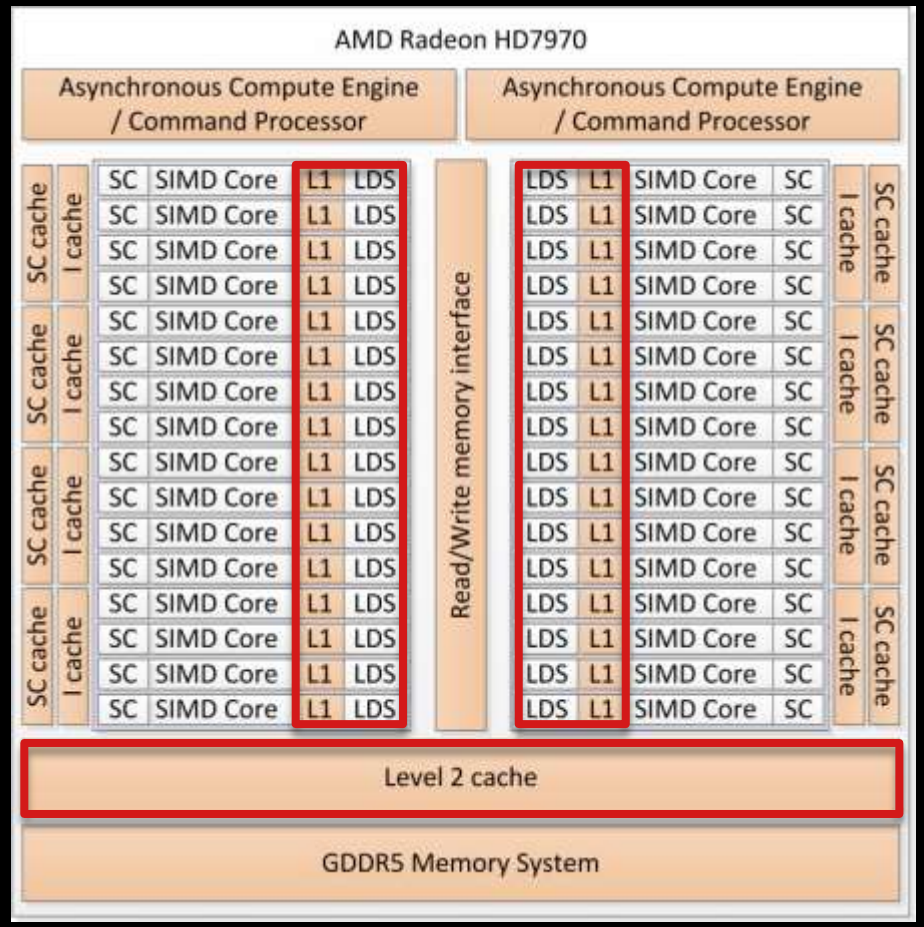
SOME LESSONS FROM THIS

- Modern GPUs are not as different from CPUs as marking departments often like to claim
  - The differences are absolutely NOT core count
  - Thread count, is a different story

- The SPMD-on-SIMD or SIMT mapping is almost entirely a tool chain construct
  - It may have some hardware acceleration
  - In general it maps to traditional vector units

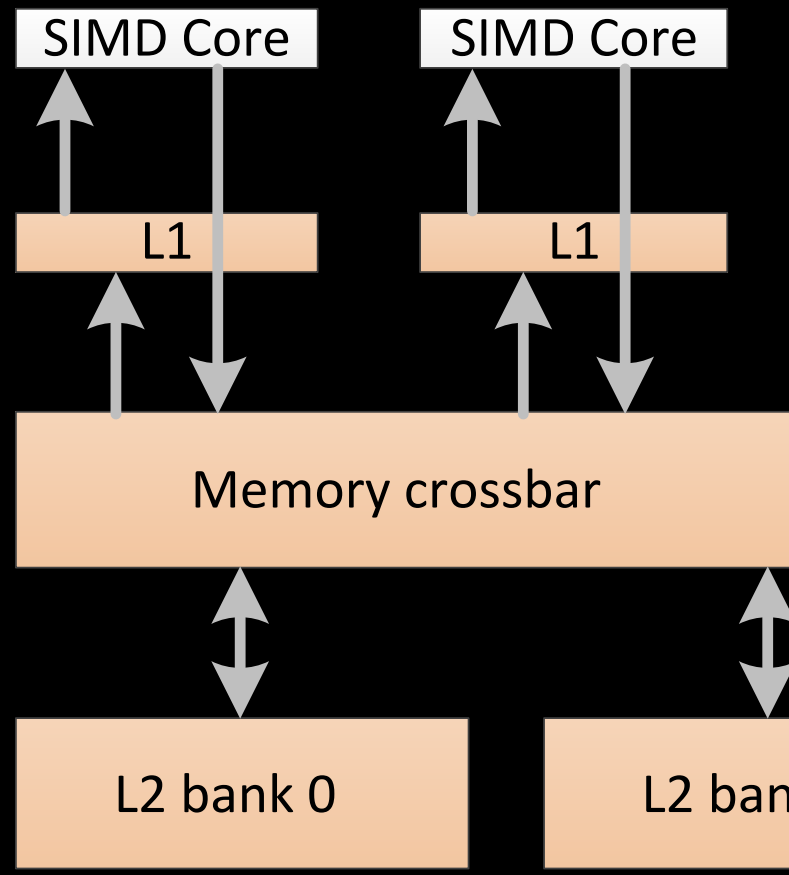
THE CACHE HIERARCHY

- Up to 512kB of L2 cache
  - Fully read/write
  - Coherent across the device
  - Associated with the memory interfaces
- 16kB of L1 cache per core
  - Fully read/write
  - Write through
  - Relaxed consistency
- Local data share
  - The __local memory in OpenCL
  - Program controlled scratchpad memory
  - Allocations are shared across multiple work-items in an OpenCL work-group
  - 64kB/core
- Note that there is also 256kB of registers per core



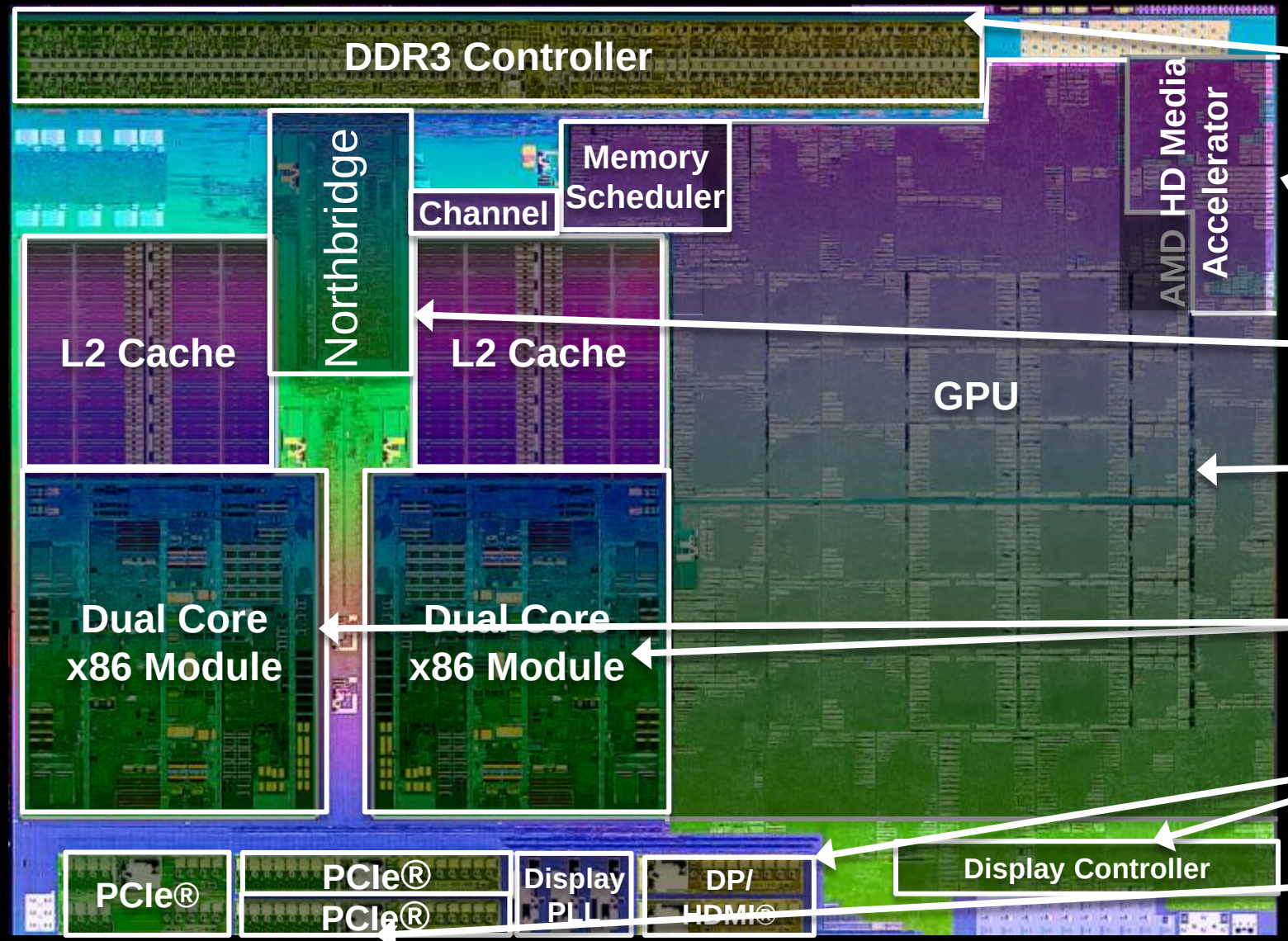
MEMORY CONSISTENCY

- Write through L1
  - Each write will commit to L2 in order
  - Dirty masks ensure merging on write to L2
  - Partially clean lines will be evicted from L1 to force a merge into a full cache line from L2
  - Fully dirty lines will not evict so the next read will be directly from L1
- Reads will read from L1 if data is available
  - Must be forced to read from L2
- Implementing consistent memory operations requires the use of two primary functions
  - Setting the GLC (globally coherent) bit on the read instruction to invalidate the L1 line and read from L2
  - Use the S_WAITCNT instruction to wait for previous memory accesses to have completed



“TRINITY” APU FLOORPLAN

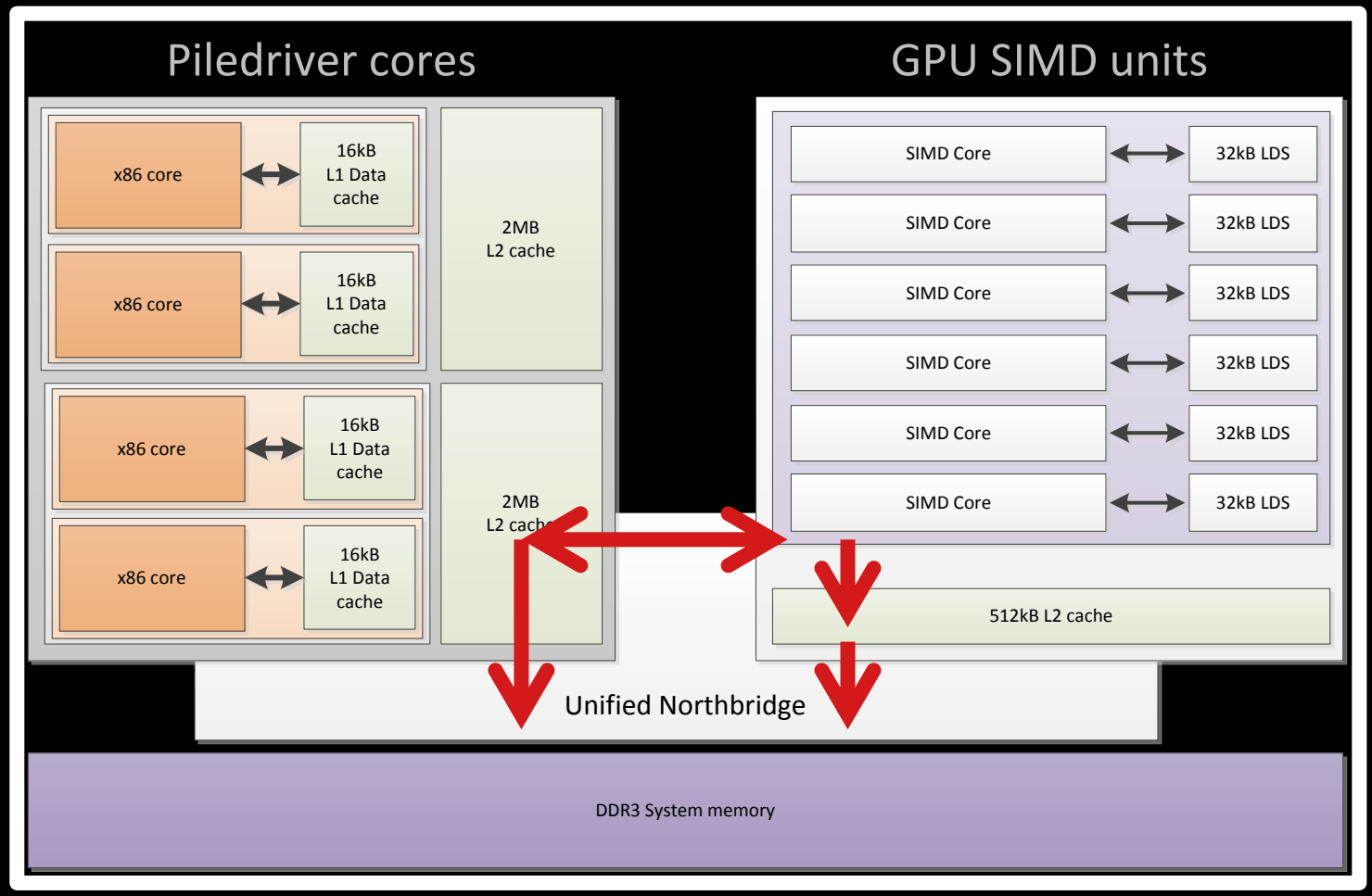
32nm SOI, 246mm2, 1.303BN TRANSISTORS



- *Dual Channel DDR3 Memory Controller*
- *AMD HD Media Accelerator (UVD, AMD Accelerated Video Converter)*
- *Unified Northbridge*
- *AMD Radeon™ GPU*
- *Up to 4 “Piledriver” Cores with total 4MB L2*
- *HDMI, DisplayPort 1.2, DVI controllers*
- *PCI Express® I/O — 24 lanes, optional digital display interfaces*

THE TRINITY APU MEMORY SYSTEM

- CPU and GPU both have direct paths to northbridge
- The GPU has a second route
  - The Fusion Control Link
  - Allows GPU access to x86 memory space
  - Allows interaction with the CPU caches
  - Supports platform coherent memory
  - Coherency switches from GPU L2 to CPU L2 for this path
- Memory allocations are marked
  - Switch between paths at the page level

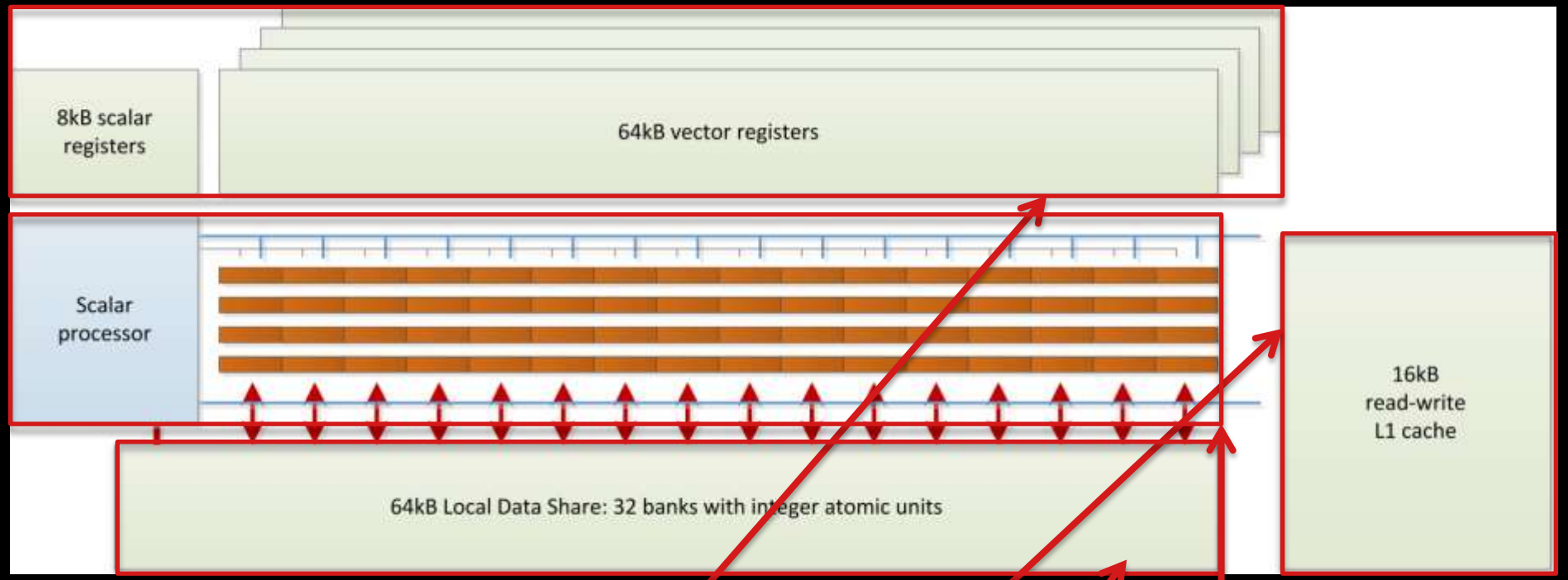


Shared Virtual Memory

EXPANDING THE MEMORY SYSTEM - HUMA

- GCN-based devices are more flexible
  - Start to look like CPUs with few obvious shortcomings
- The Trinity APU integrates GPU and CPU cores very closely
  - It can share memory consistently
  - However, that shared memory must be addressed physically
  - It is pinned by the driver – this is limited
- 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?



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

IOMMUV2

- The final step is to coherent, virtual and pageable access to system memory from the GPU's memory controller

- The GPU needs to:
  - Use a virtual x86 address
  - Find that that address in the TLB
  - If the address is not in the TLB, read the page tables to find it
  - If the page is not in memory at all, ask that it be moved in and wait for completion

- The latest discrete GPUs can do this, and Trinity has the beginnings with the IOMMU (input output memory management unit) version 2
  - Although Trinity's GPU cores are an earlier generation and cannot make use of it

USE CASES FOR THIS ARE FAIRLY OBVIOUS

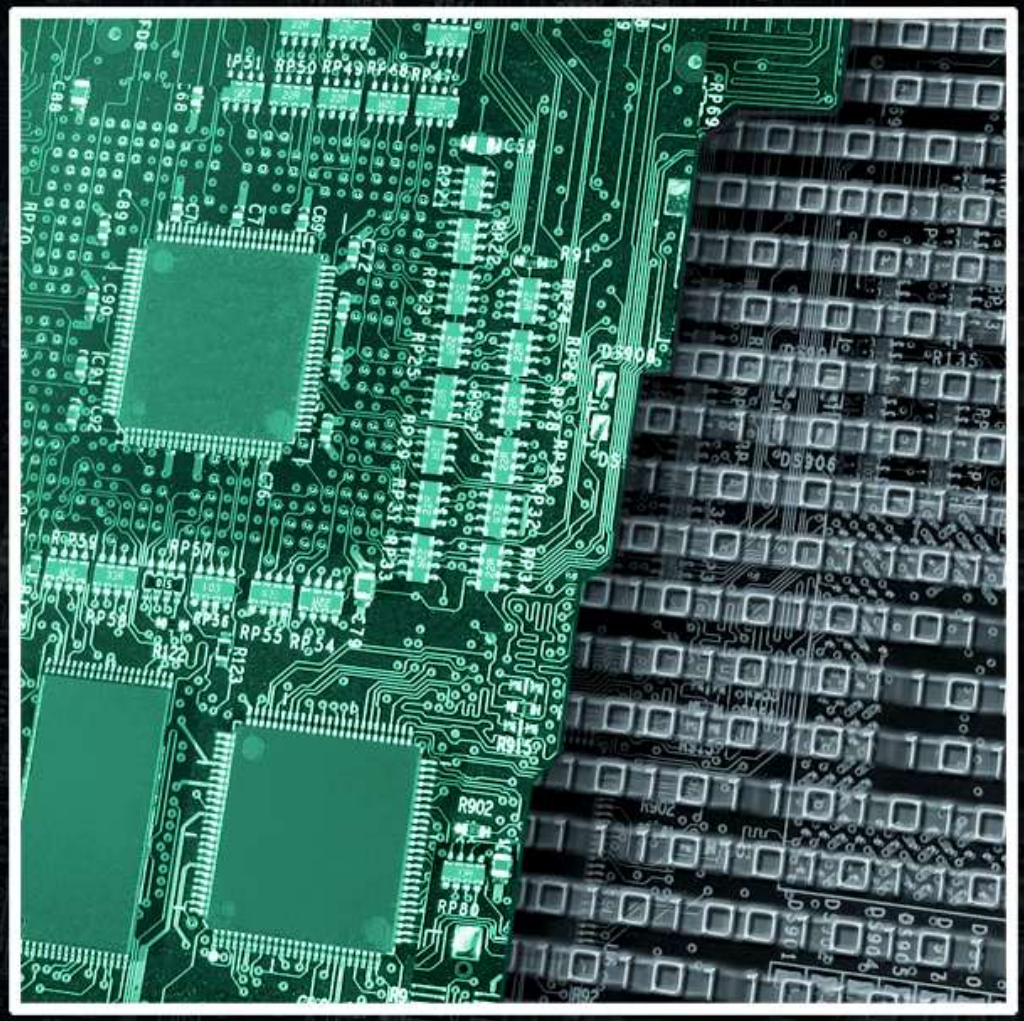
AMD

- 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…

Architected Access

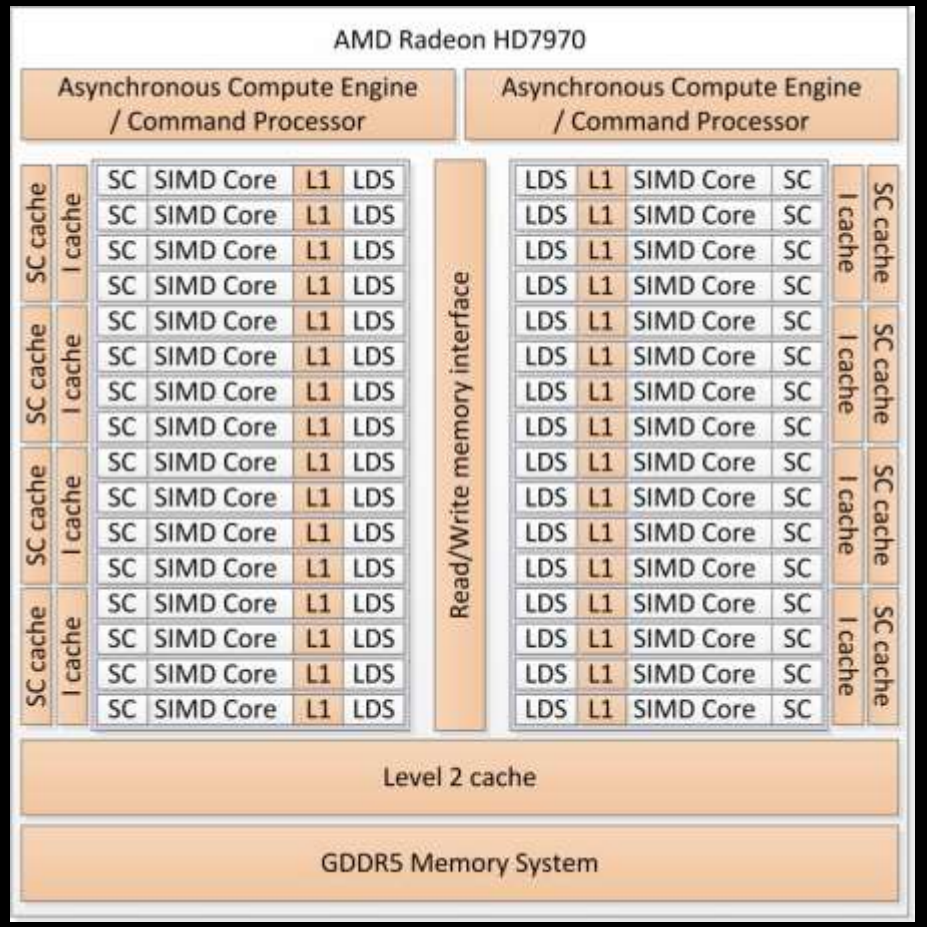
HETEROGENEOUS SYSTEM ARCHITECTURE – AN OPEN PLATFORM

- 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 has been formed



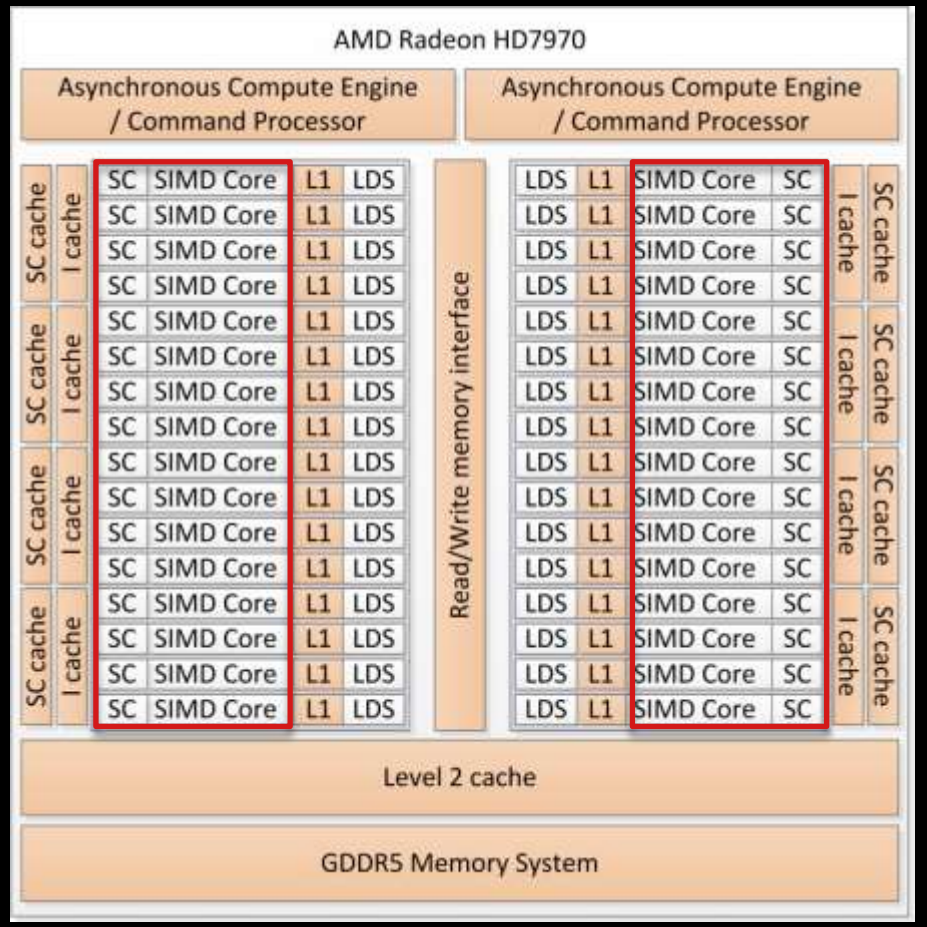
ARCHITECTED INTERFACES

- 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



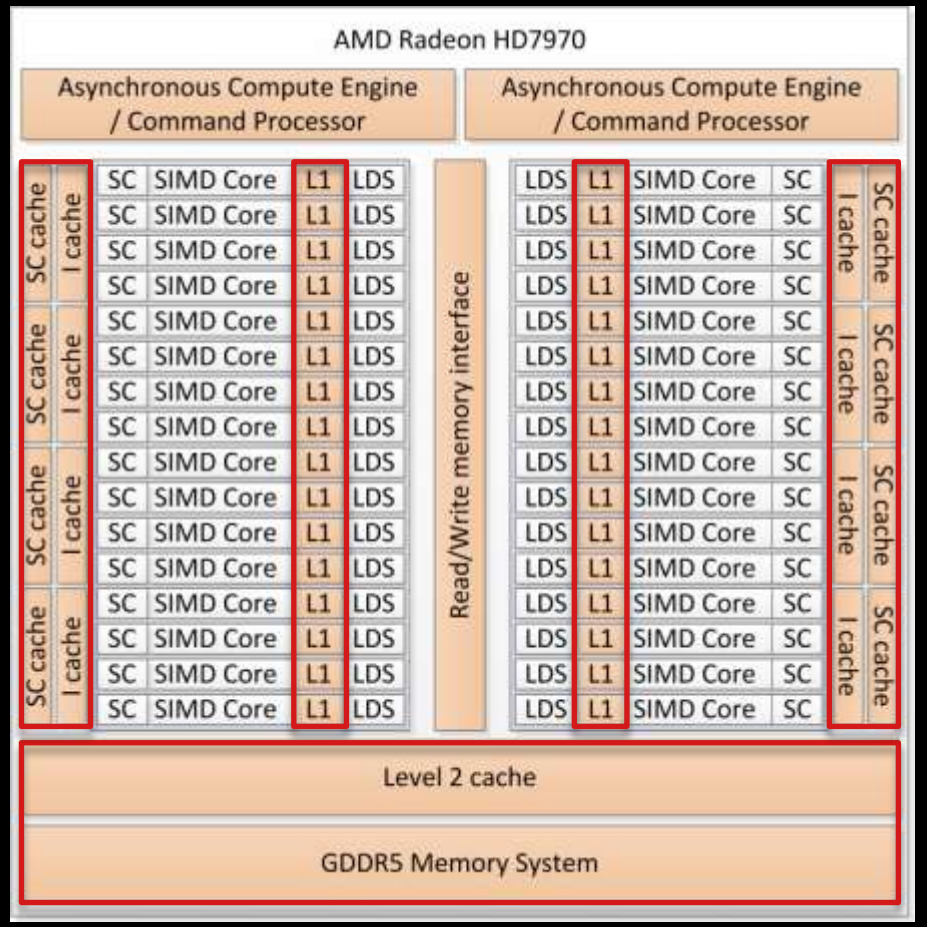
HSA INTERMEDIATE LAYER - HSAIL

- 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



HSA MEMORY MODEL

- A strict memory model allows us to reason about correctness of communicating processes
- This sort of strengthening of the memory model allows for predictable locks, lock free data structures and similar concepts
  - Work item -> work item communication
  - Pipelines
  - Locking of shared data by concurrent work items
- It also provides a stronger basis for academic research
  - While relying on OpenCL concurrency has never been portable, adding the HSA memory model would at least make the communication guarantees reliable

HSAIL 0.95 PROGRAMMERS REFERENCE MANUAL NOW PUBLIC

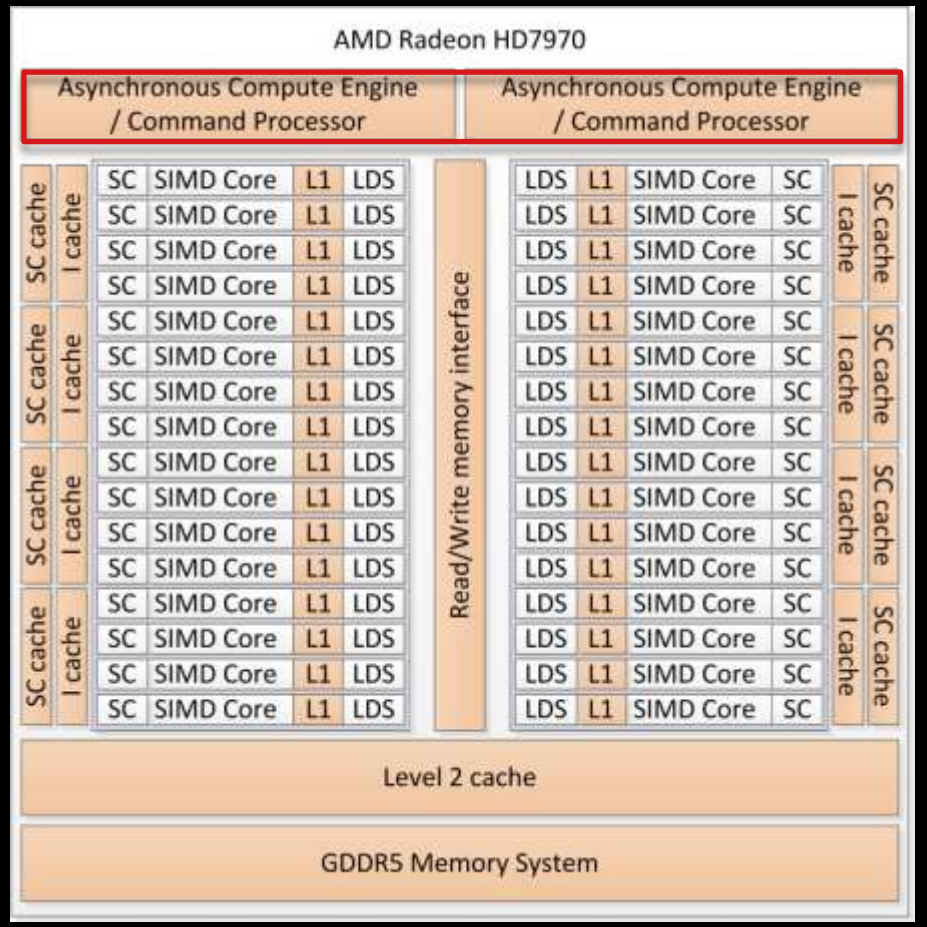
AMD

- Available at hsafoundation.com
- Announced on May 29th
- Describes:
  - The RISC-like virtual ISA
  - Binary format
  - Memory model

```
version 1:0:$full:$small;

function &get_global_id(arg_u32 %ret_val) (arg_u32 %arg_val0);

function &abort() ();

kernel &__OpenCL_vec_add_kernel(
    kernarg_u32 %arg_val0,
    kernarg_u32 %arg_val1,
    kernarg_u32 %arg_val2,
    kernarg_u32 %arg_val3)
{
@__OpenCL_vec_add_kernel_entry:
// BB#0:                                // %entry
    ld_kernarg_u32    $s0, [%arg_val3];
    workitemabsid_u32 $s1, 0;
    cmp_lt_b1_u32     $c0, $s1, $s0;
    ld_kernarg_u32    $s0, [%arg_val2];
    ld_kernarg_u32    $s2, [%arg_val1];
    ld_kernarg_u32    $s3, [%arg_val0];
    cbr    $c0, @BB0_2;
    brn    @BB0_1;
@BB0_1:                                // %if.end
    ret;
@BB0_2:                                // %if.then
    shl_u32     $s1, $s1, 2;
    add_u32     $s2, $s2, $s1;
    ld_global_f32    $s2, [$s2];
    add_u32     $s3, $s3, $s1;
    ld_global_f32    $s3, [$s3];
    add_f32     $s2, $s3, $s2;
    add_u32     $s0, $s0, $s1;
    st_global_f32    $s2, [$s0];
    brn    @BB0_1;
};
```

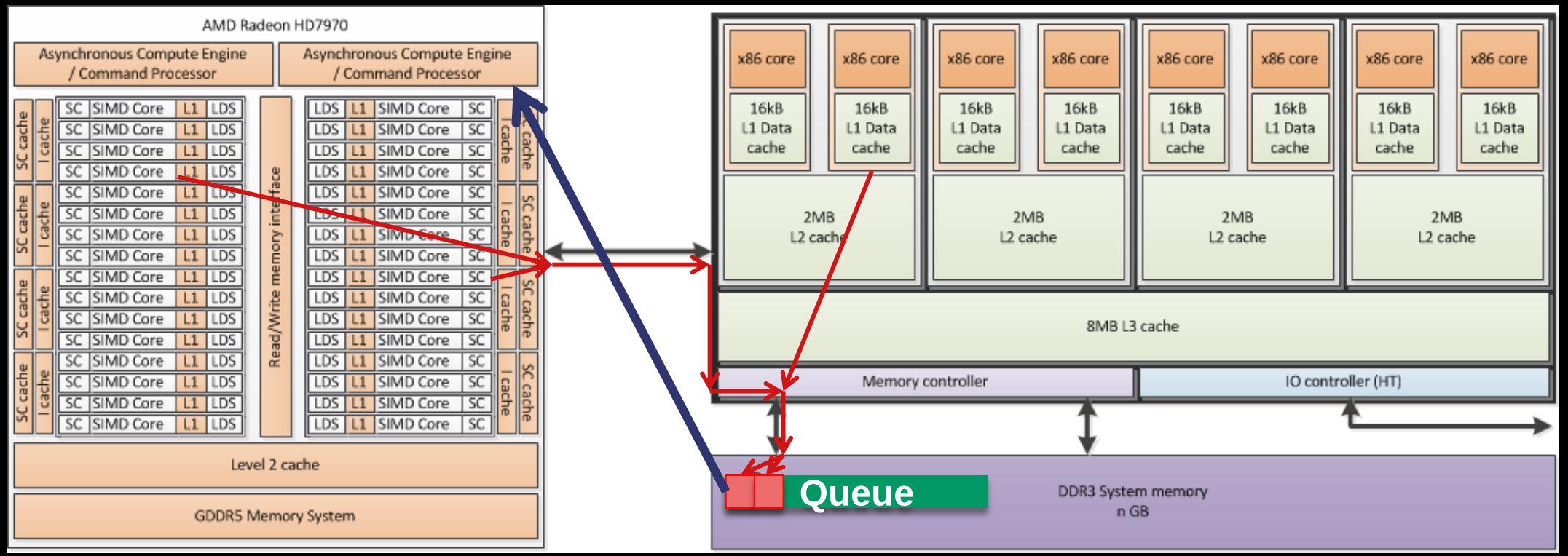
ARCHITECTED QUEUING LANGUAGE

- 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



Architected Dispatch

WHAT DO THESE CAPABILITIES OFFER US?

- Combining:
  - Shared virtual memory
  - A strong memory model
  - Architected communication packets the world opens
- Offers huge power, safety and a world of opportunities
- Let’s look at a couple of immediate benefits

SECURITY IMPROVEMENTS WITH HSA

AMD

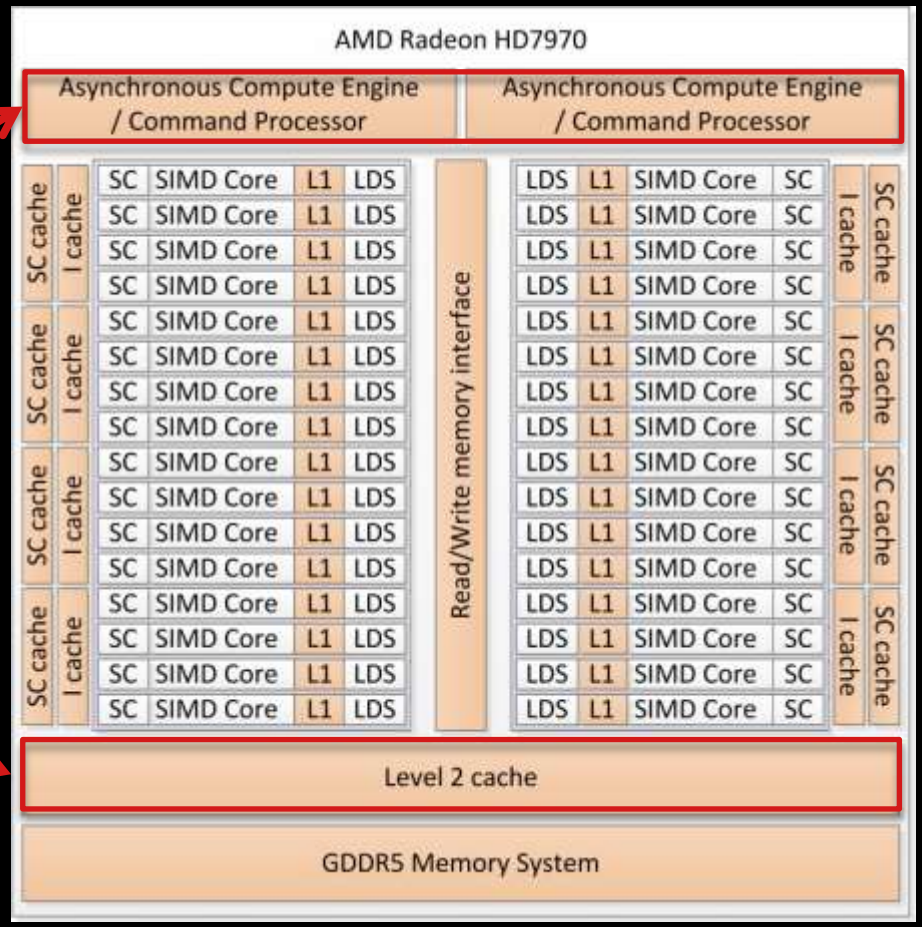
- With HSA, GPU operates in the same security infrastructure as the CPU
  - User and privileged memory
  - Read, write and execute protections by page table entry
- Internally, the GPU partitions functionality by privilege level
  - User mode compute queues can only run AQL packets
  - User mode graphics command buffers cannot write privileged registers
- HSA supports fixed time context switching, which is resistant to denial of service (DoS) attacks
  - Today's GPUs are vulnerable to denial of service attacks
    - Long or infinite shader programs
    - Full GPU reset required to restore service
  - With HSA, fair scheduling and context switching ensures a responsive system

EXPANDING SVM SCOPE TOWARDS POWERFUL DEVICE CONTROL

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

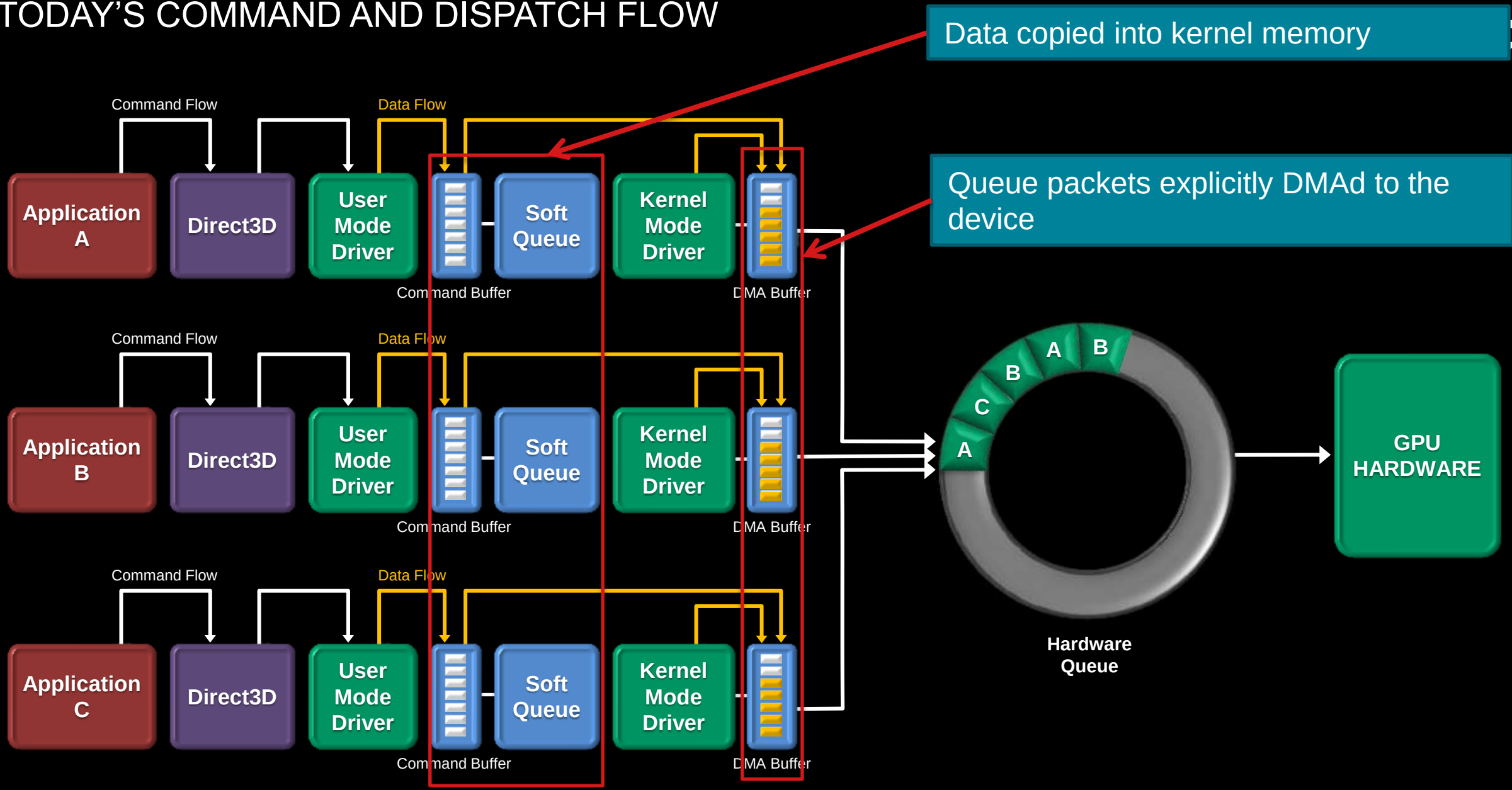
Most importantly! We need to be able to compute on the same data here.

Of course, we need to see the data here too

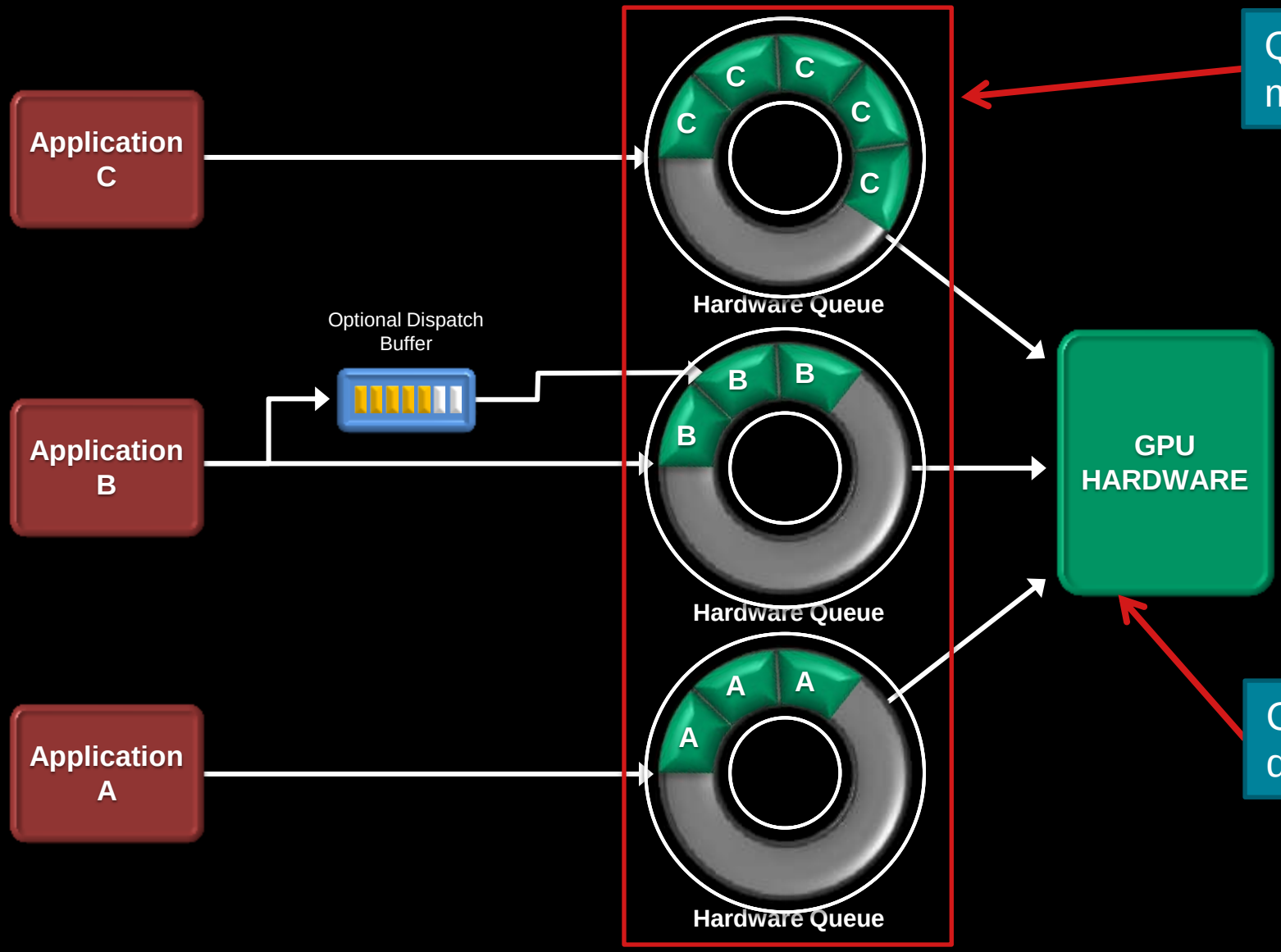


- Let's look at how GPU work dispatch works currently

TODAY'S COMMAND AND DISPATCH FLOW



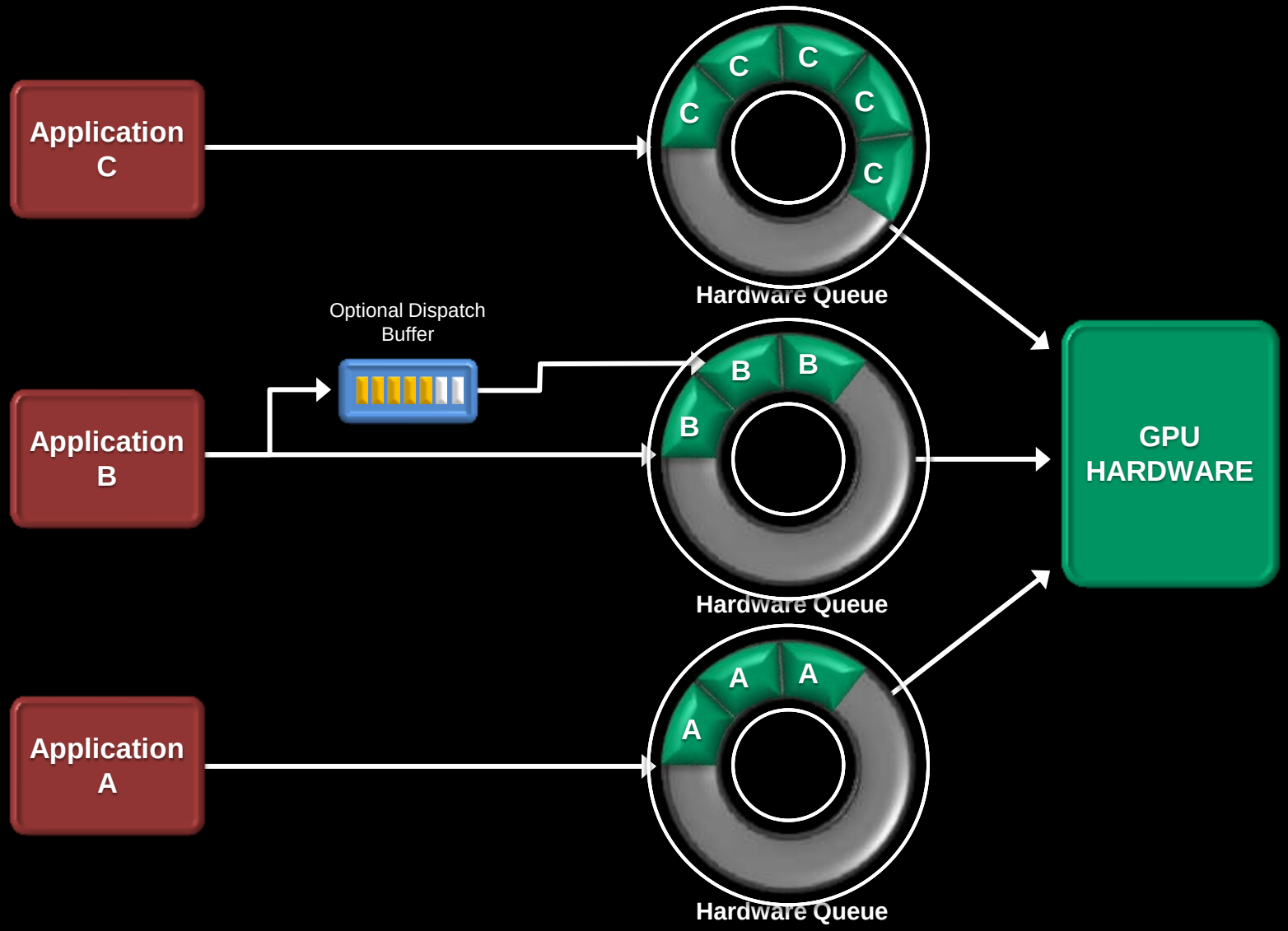
FUTURE COMMAND AND DISPATCH FLOW



Queues are in user-mode virtual memory

Command processor can read queues directly

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!

Advances do not stop…

ARCHITECTURE PROGRESSION

AMD

Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU & GPU in silicon	GPU Compute C++ support	Unified Address Space for CPU and GPU	GPU compute context switch
Unified Memory Controller	User mode schedulng	GPU uses pageable system memory via CPU pointers	GPU graphics pre-emption
Common Manufacturing Technology	Bi-Directional Power Mgmt between CPU and GPU	Fully coherent memory between CPU & GPU	Quality of Service
			Extend to Discrete GPU

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. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION