HSA Foundation

Antonio Miele

Politecnico di Milano

POLITECNICO
MILANO 1863

References

- This presentation is based on the material and slides published on the HSA foundation website:
  - http://www.hsafoundation.com/

Heterogeneous processors have proliferated – make them better

- Heterogeneous SoCs have arrived and are a tremendous advance over previous platforms
- SoCs combine CPU cores, GPU cores and other accelerators, with high bandwidth access to memory
- How do we make them even better?
  - Easier to program
  - Easier to optimize
  - Easier to load balance
  - Higher performance
  - Lower power
- HSA unites accelerators architecturally
- Early focus on the GPU compute accelerator, but HSA will go well beyond the GPU

HSA foundation

- Founded in June 2012
- Developing a new platform for heterogeneous systems
- www.hsafoundation.com
- Specifications under development in working groups to define the platform
- Membership consists of 43 companies and 16 universities
- Adding 1-2 new members each month

HSA consortium

Founders

AMD ARM Imagination MEDIATEK
QUALCOMM SAMSUNG TEXAS INSTRUMENTS

Promoters

LG Electronics

Supporters

Arteris FABRICENGINE KISHONTI LAWRENCE LIVERMORE NATIONAL LABORATORY allinea Argonne
codeplay MULTICORE WARE Linaro Sandia National Laboratories SUSE We adapt. You succeed. OAK RIDGE National Laboratory Stream Computing Performance Engineers

Contributors

ANALOG DEVICES apical BROADCOM. CANONICAL CEVA ETRI 한국전자통신연구원 Electronics and Telecommunications Research Institute
ITRI Industrial Technology Research Institute MARVELL mobica DMP SONICS SONY make.believe Swarm64 HUAWEI
SYNOPSYS tensilica TOSHIBA Leading Innovation >>> ubuntu VIA VIVANTE S3 GRAPHICS ORACLE

Academic

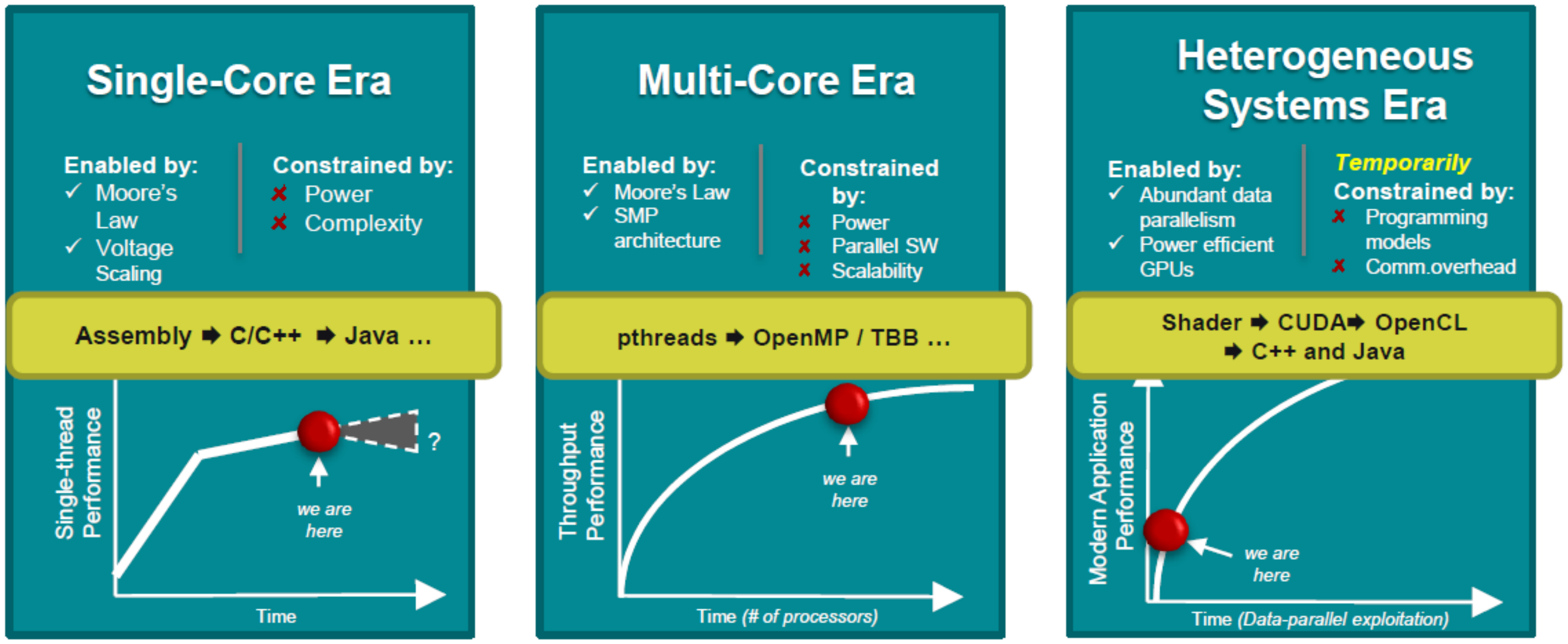
NTHU Programming Language Lab NTHU System Software Lab NMAMIT University of BRISTOL THE UNIVERSITY of EDINBURGH informatics ILLINOIS COMPUTER SCIENCE
清華大學 Tsinghua University TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE UNIVERSITY of MISSISSIPPI ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

HSA goals

- To enable power-efficient performance
- To improve programmability of heterogeneous processors
- To increase the portability of code across processors and platforms
- To increase the pervasiveness of heterogeneous solutions throughout the industry

Paradigm shift

- Inflection in processor design and programming



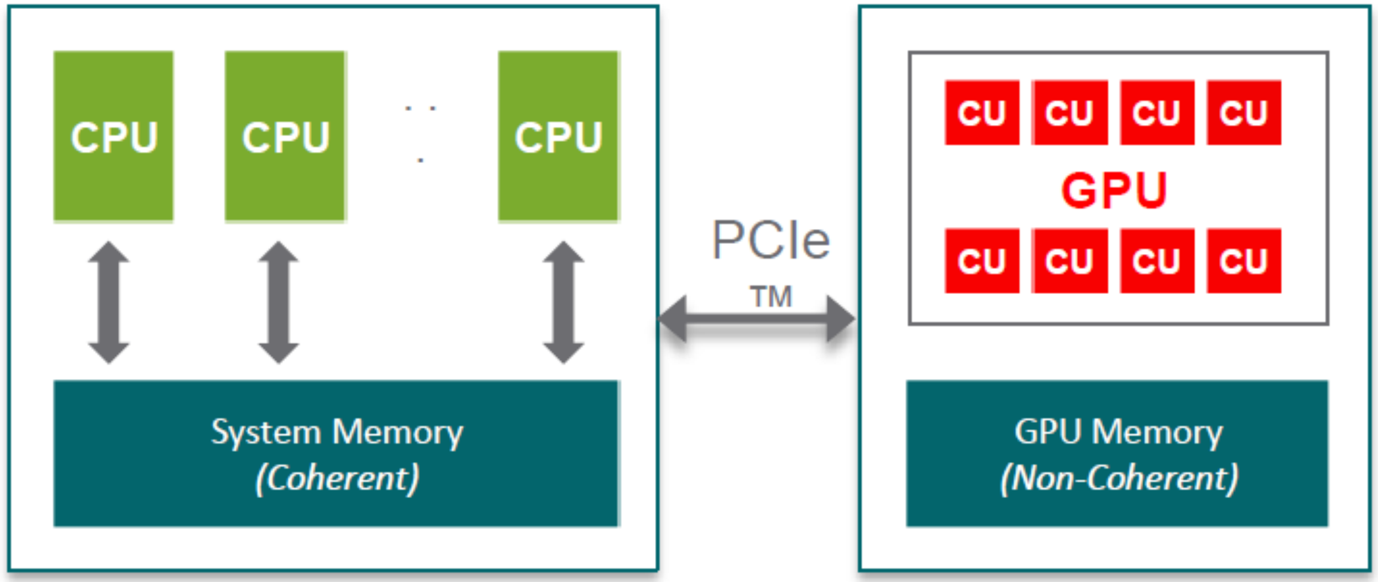
Key features of HSA

- hUMA – Heterogeneous Unified Memory Architecture
- hQ – Heterogeneous Queuing
- HSAIL – HSA Intermediate Language

Key features of HSA

- hUMA – Heterogeneous Unified Memory Architecture
- hQ – Heterogeneous Queuing
- HSAIL – HSA Intermediate Language

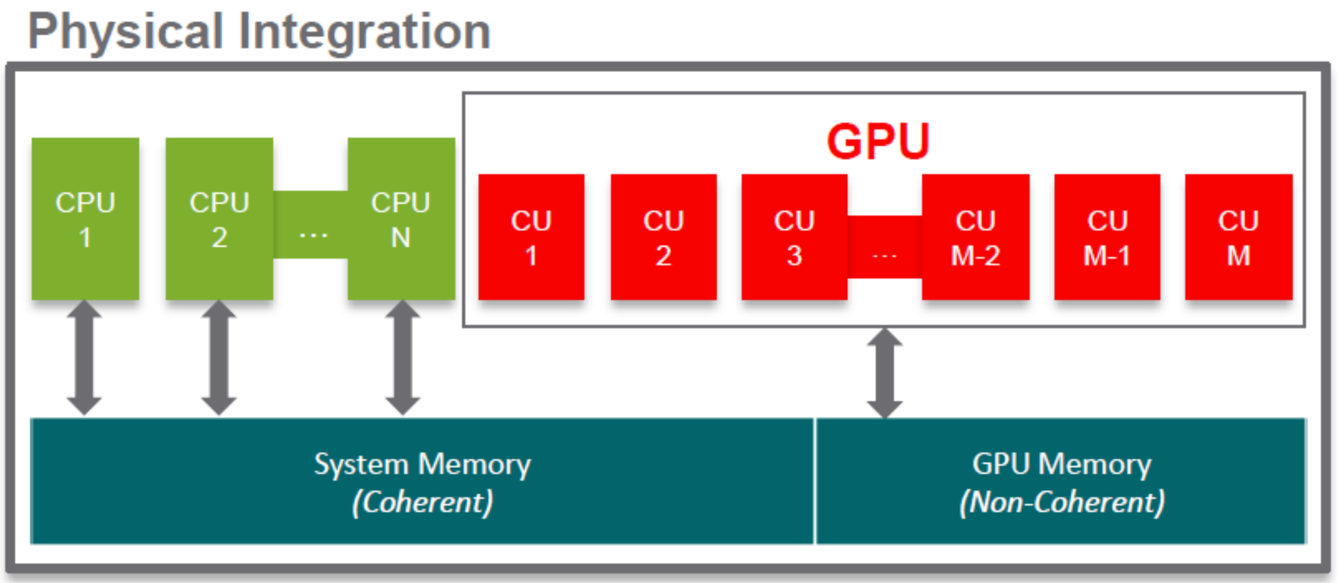
Legacy GPU compute



- Multiple memory pools
- Multiple address spaces
  - No pointer-based data structures
- Explicit data copying across PCIe
  - High latency
  - Low bandwidth
- High overhead dispatch

- Need lots of compute on GPU to amortize copy overhead
- Very limited GPU memory capacity
- Dual source development
- Proprietary environments
- Expert programmers only

Existing APUs and SoCs



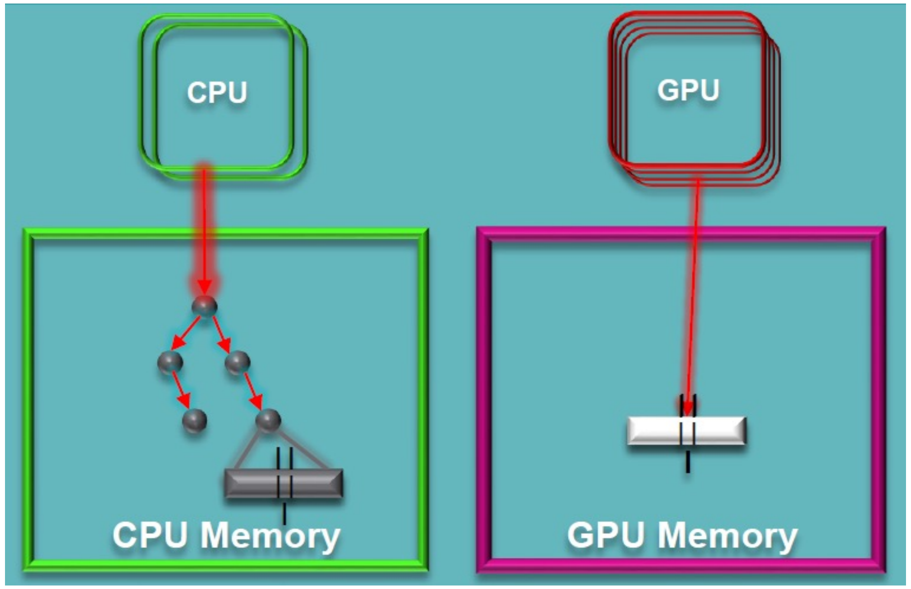
APU = Accelerated Processing Unit (i.e., a SoC containing also a GPU)

- Physical integration of GPUs and CPUs
- Data copies on an internal bus
- Two memory pools remain
- Still queue through the OS
- Still requires expert programmers

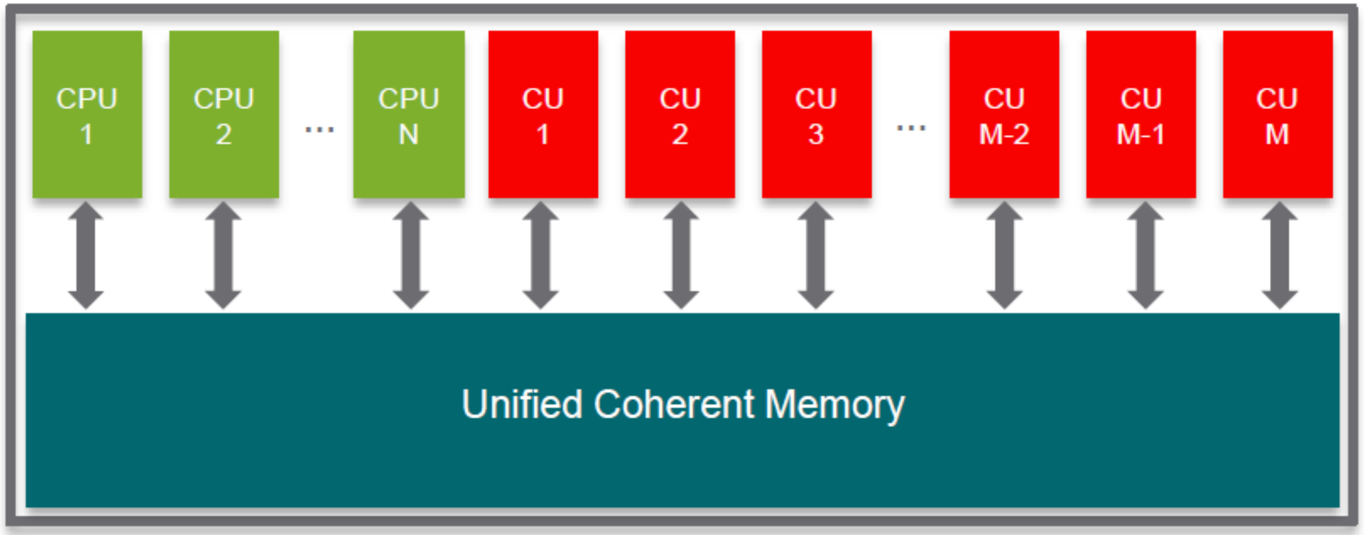
- FPGAs and DSPs have the same issues

Existing APUs and SoCs

- CPU and GPU still have separate memories for the programmer (different virtual memory spaces)
  1. CPU explicitly copies data to GPU memory
  2. GPU executes computation
  3. CPU explicitly copies results back to its own memory

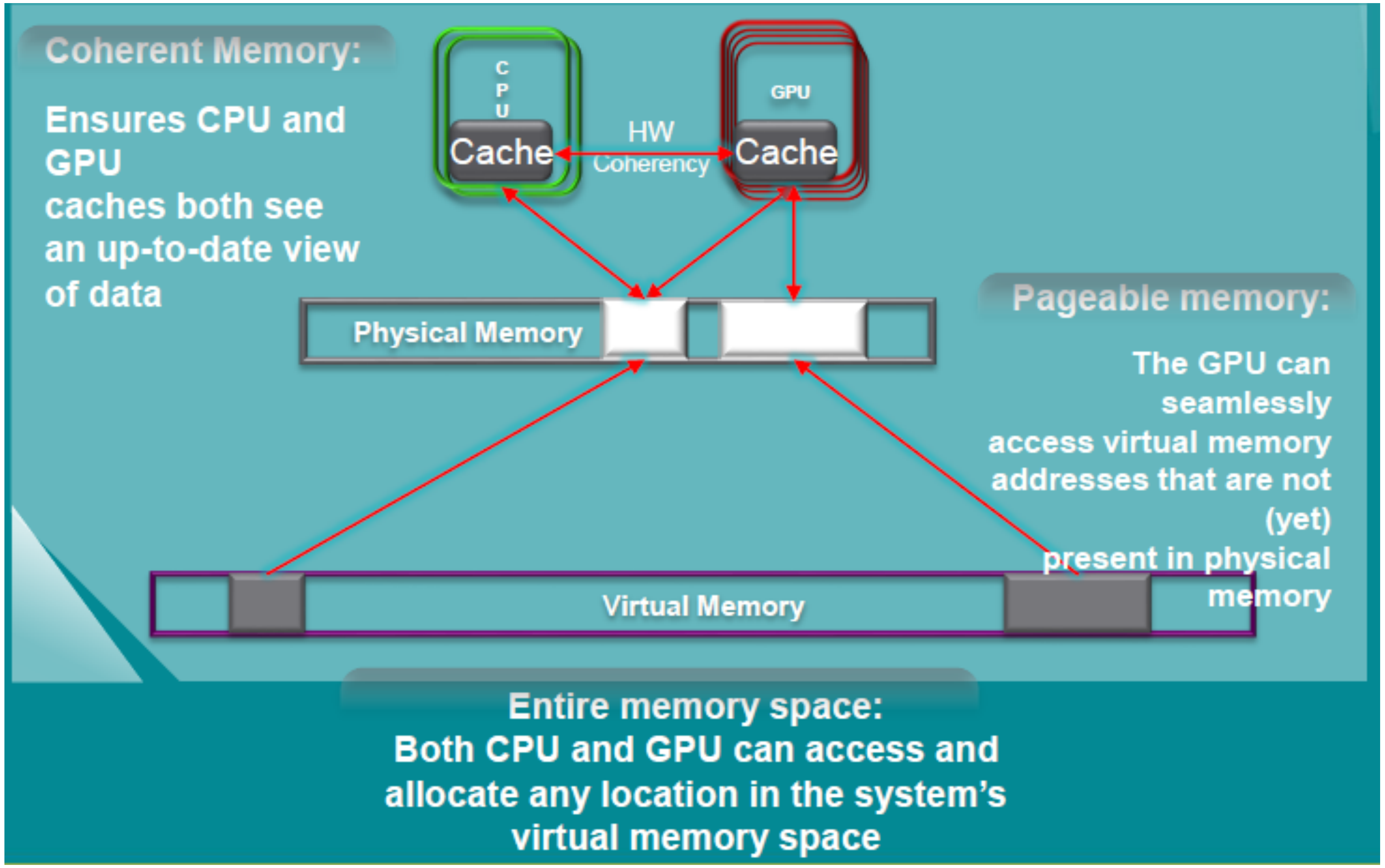


An HSA enabled SoC



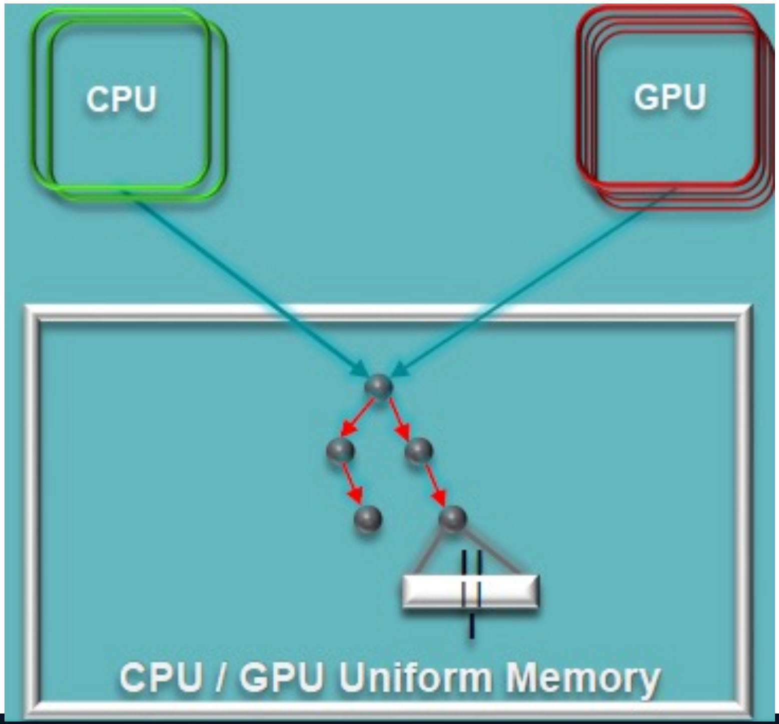
- Unified Coherent Memory enables data sharing across all processors
  - Enabling the usage of pointers
  - Not explicit data transfer -> values move on demand
  - Pageable virtual addresses for GPUs -> no GPU capacity constraints
- Processors architected to operate cooperatively
- Designed to enable the application to run on different processors at different times

Unified coherent memory

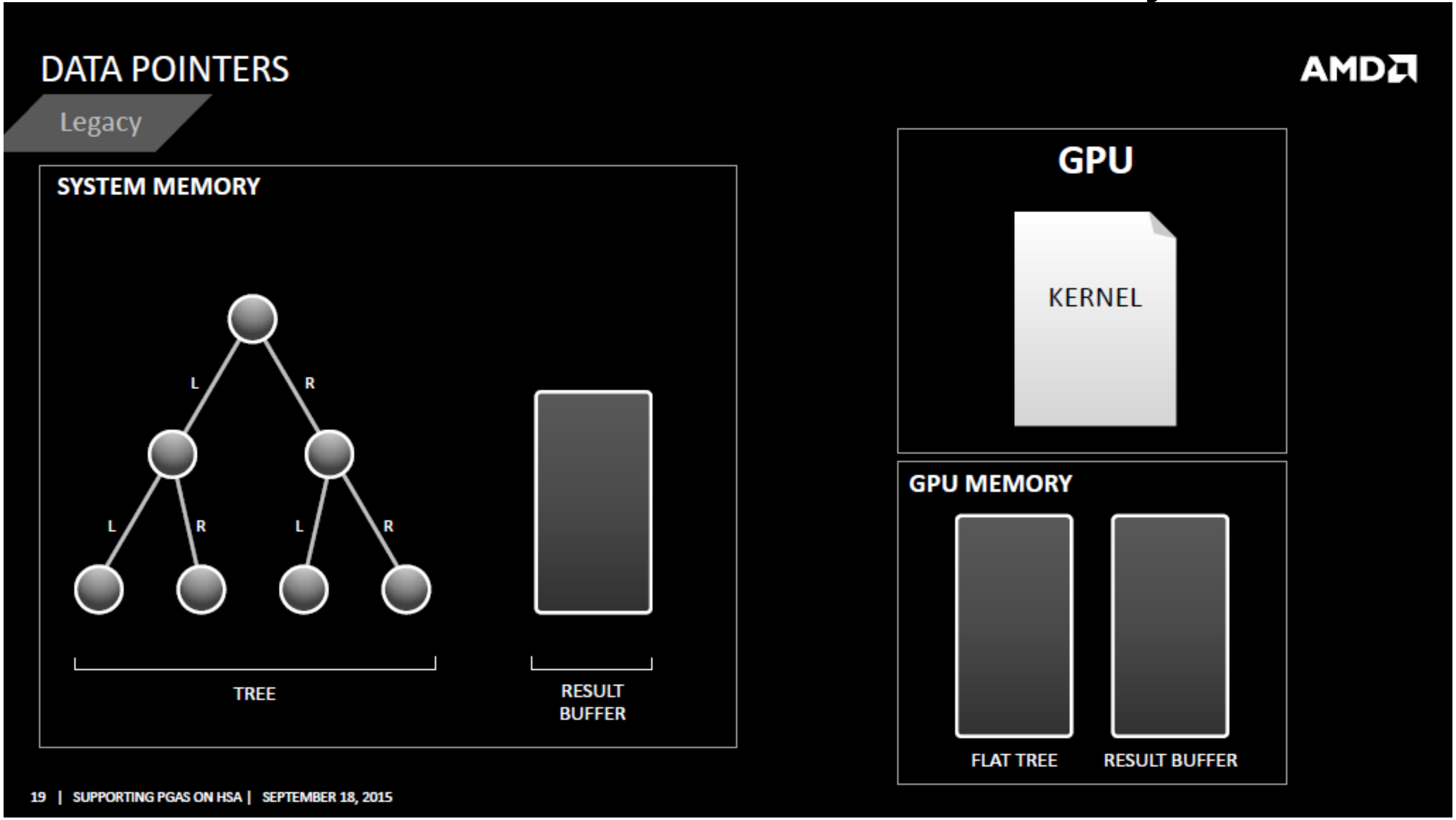

Coherent Memory:
Ensures CPU and GPU caches both see an up-to-date view of data
C P U
Cache
HW Coherency
GPU
Cache
Physical Memory
Pageable memory:
The GPU can seamlessly access virtual memory addresses that are not (yet) present in physical memory
Virtual Memory
Entire memory space:
Both CPU and GPU can access and allocate any location in the system's virtual memory space

Unified coherent memory

- CPU and GPU have a unified virtual memory spaces
  1. CPU simply passes a pointer to GPU
  2. GPU executes computation
  3. CPU can read the results directly – no explicit copy need!



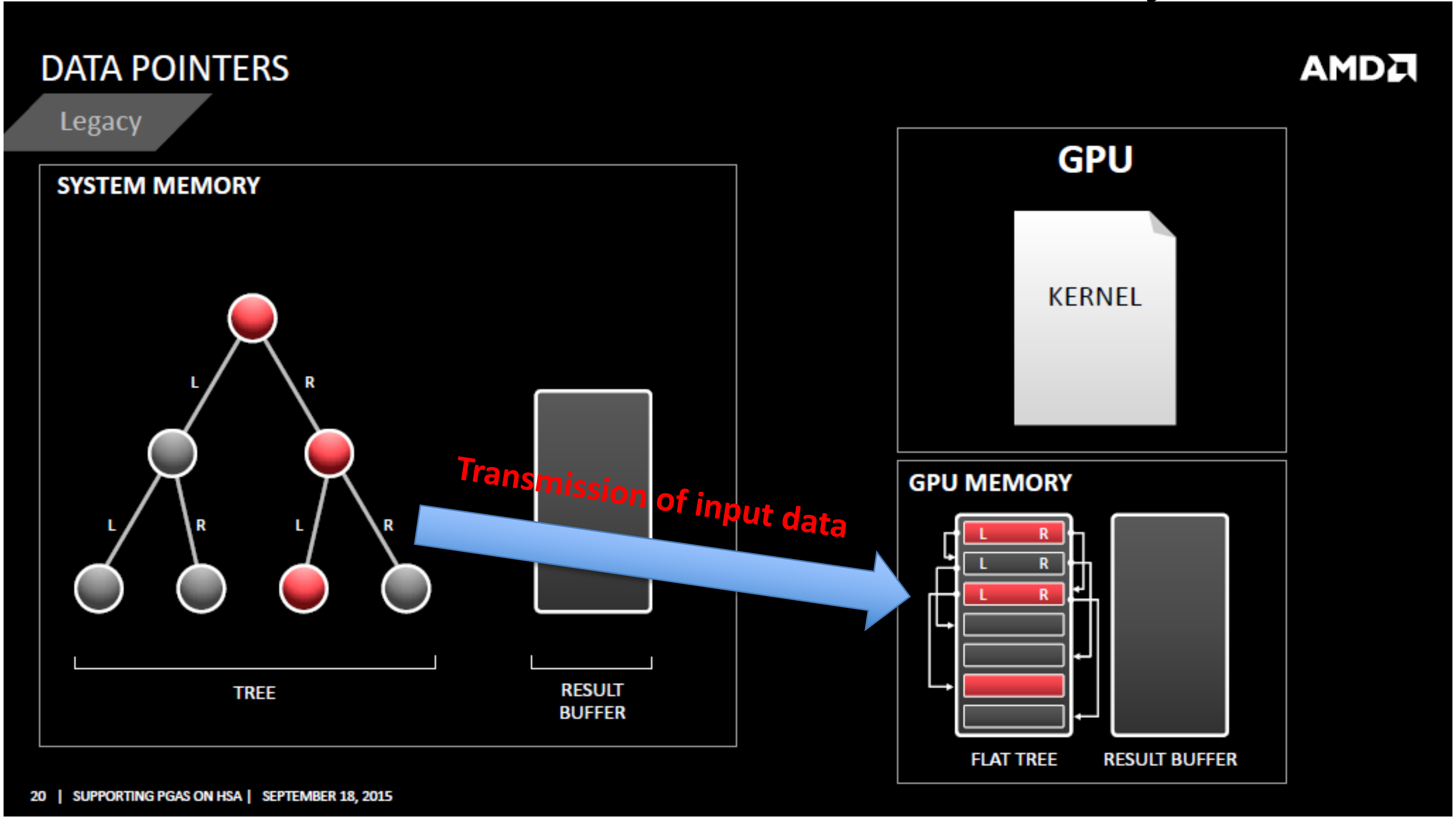
Unified coherent memory



DATA POINTERS

Legacy

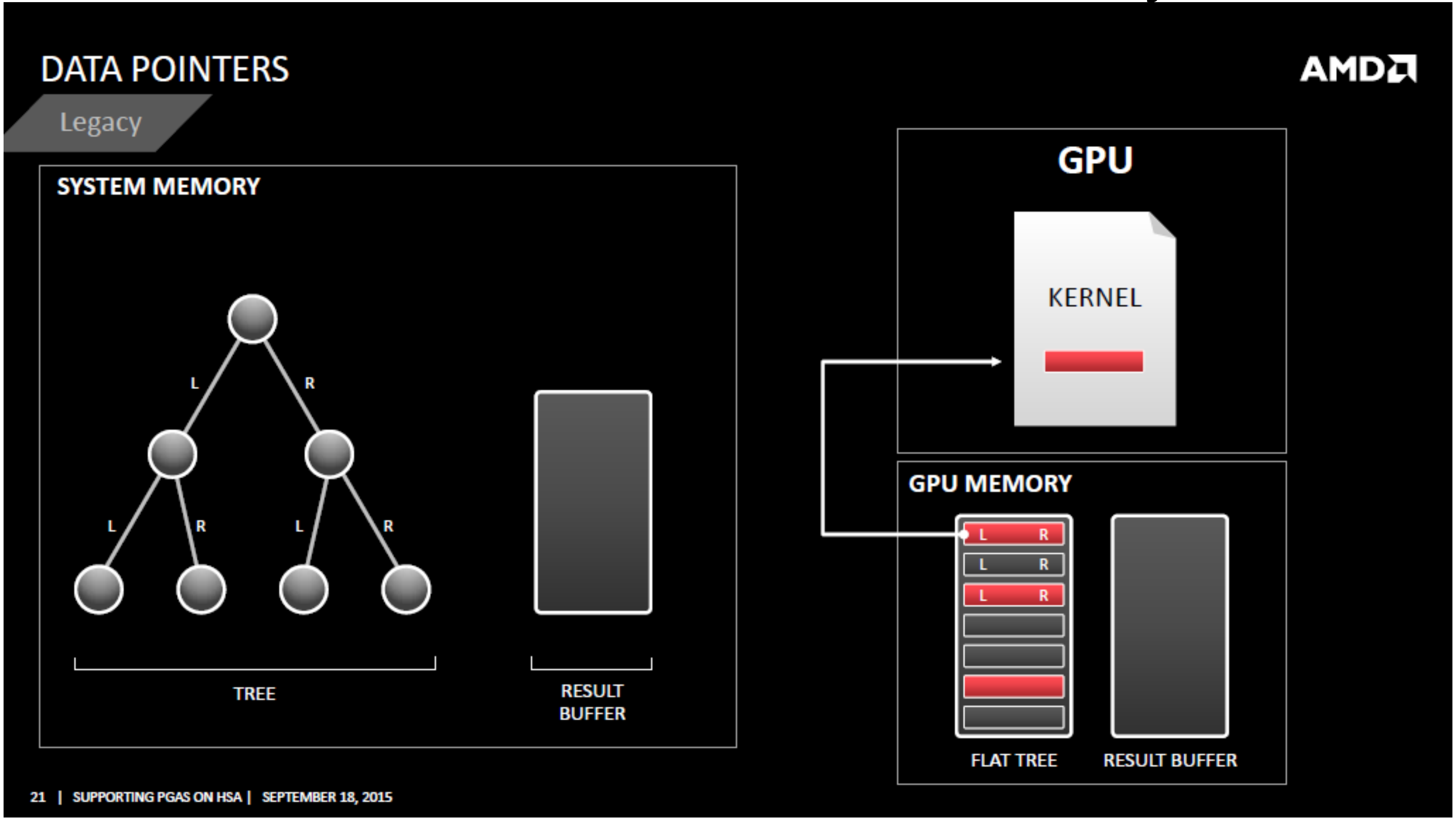
Unified coherent memory



DATA POINTERS

Legacy

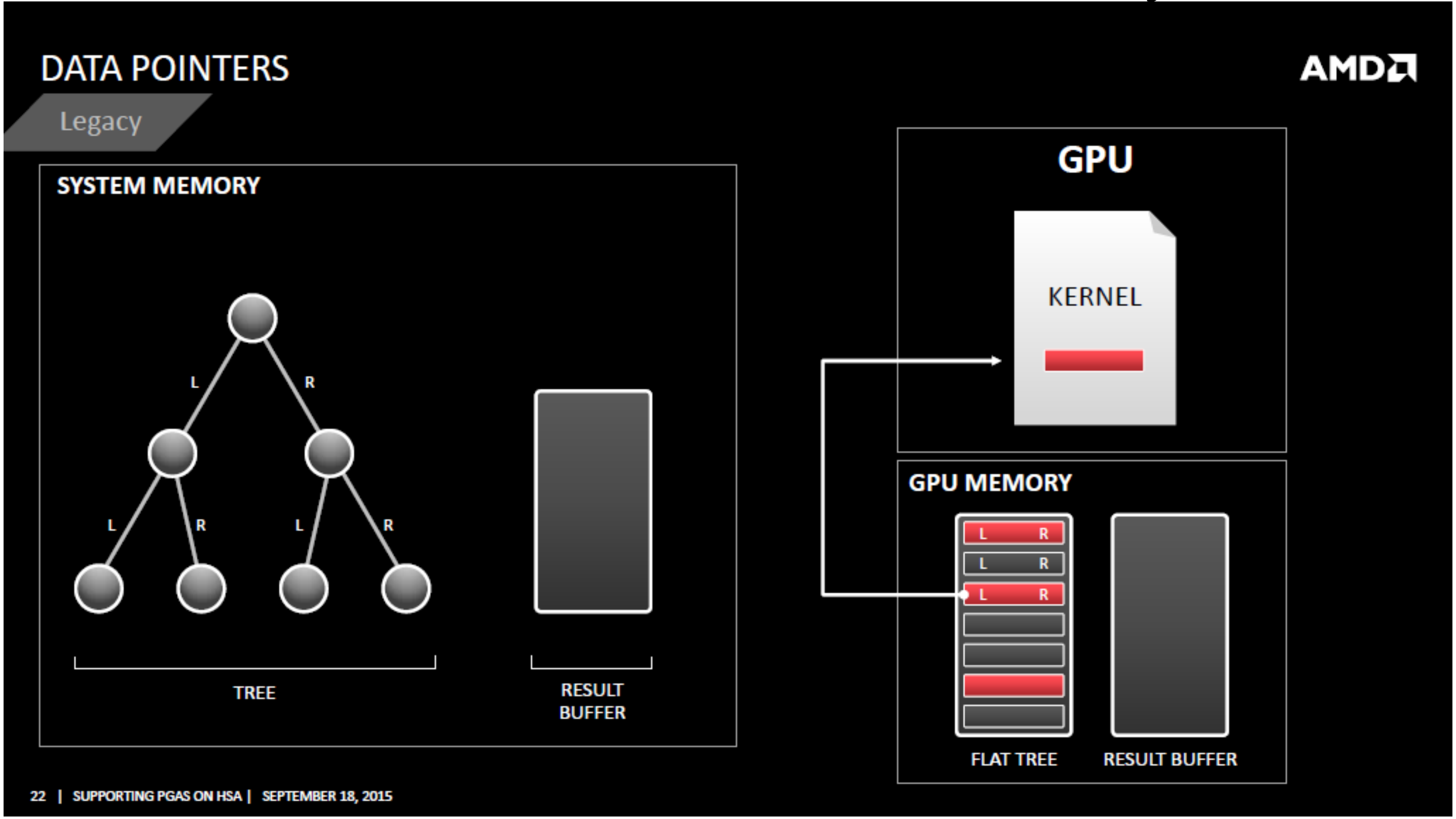
Unified coherent memory


DATA POINTERS

Legacy

21 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

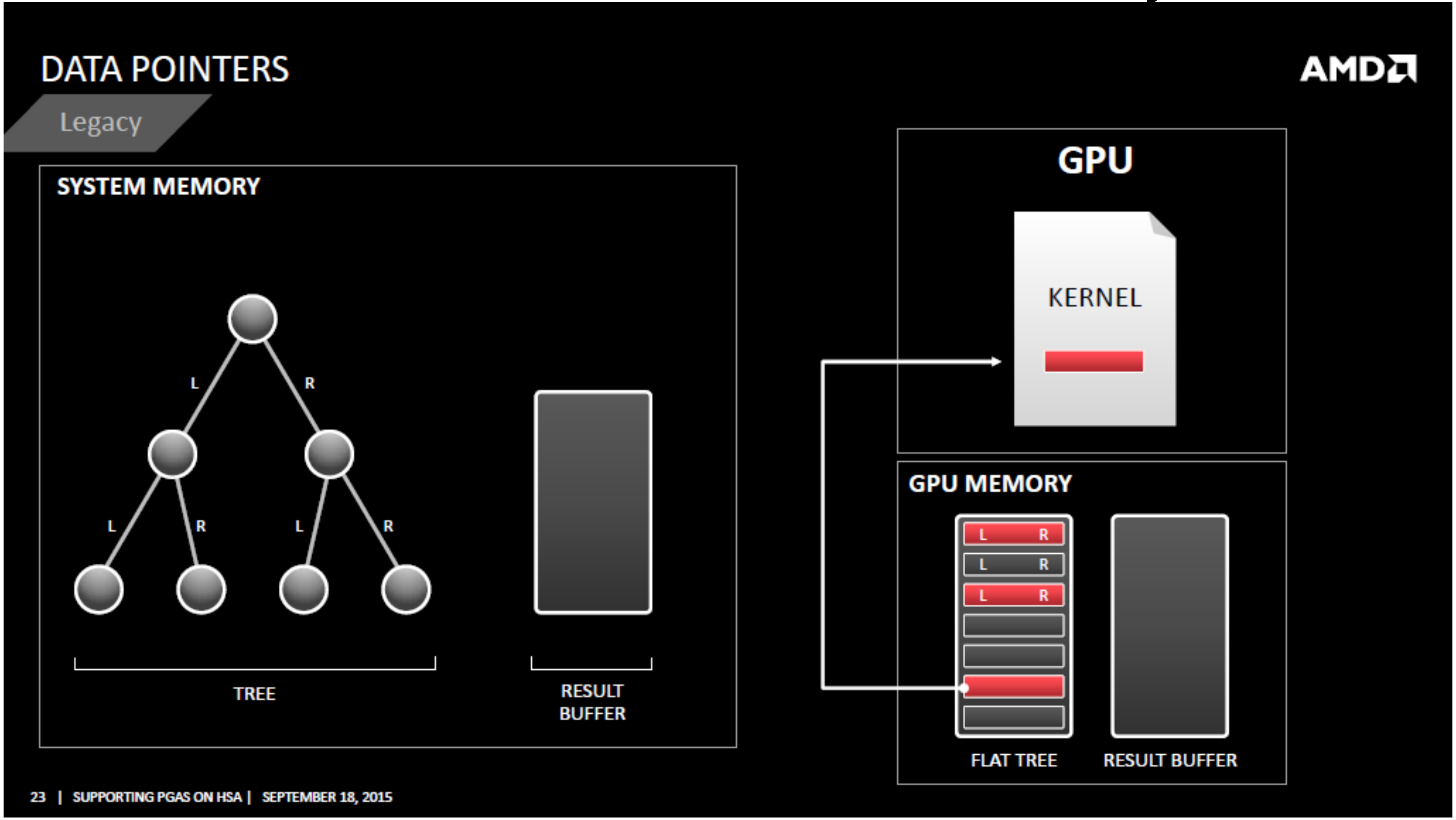
Unified coherent memory



DATA POINTERS

Legacy

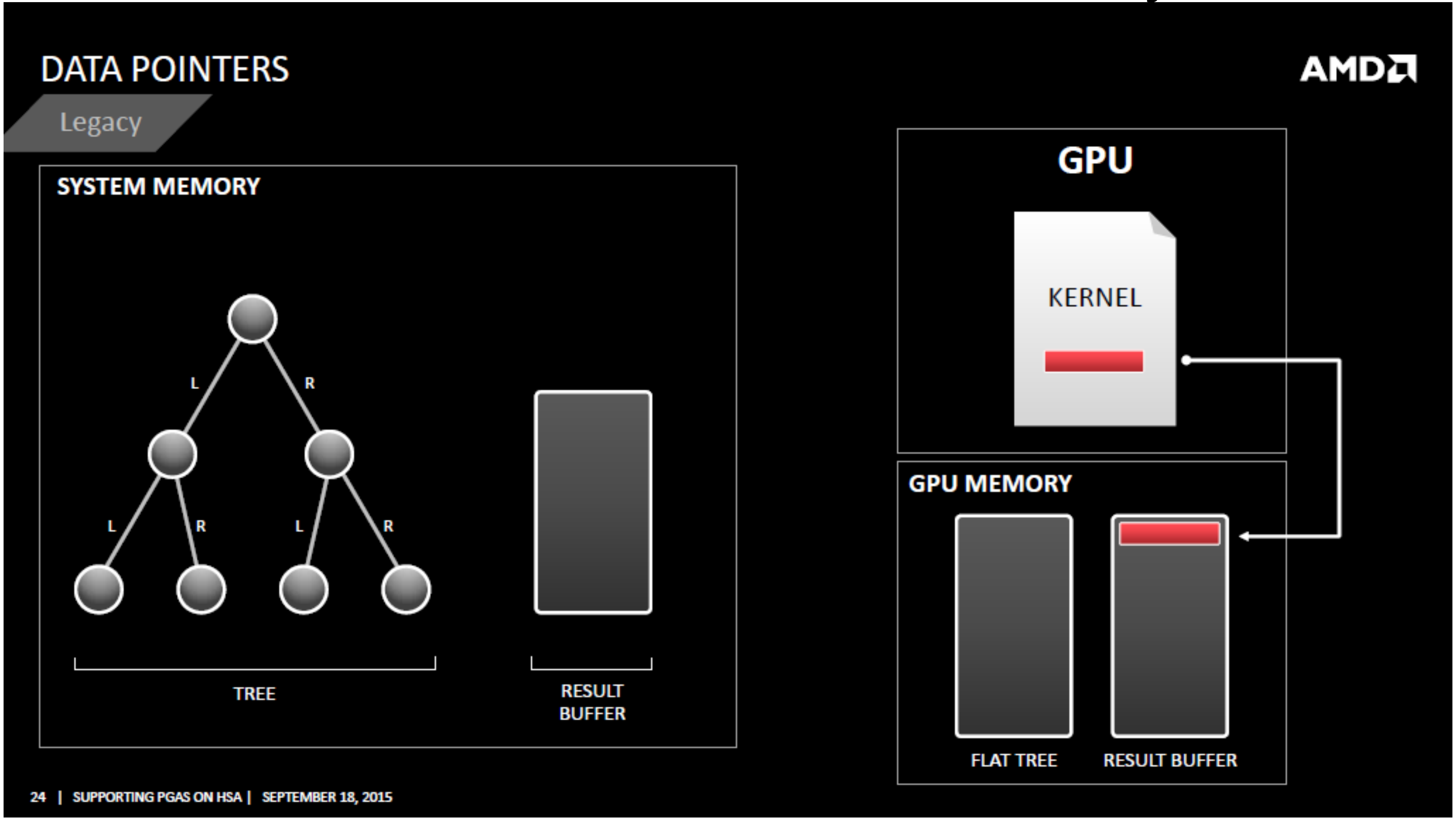
Unified coherent memory


DATA POINTERS

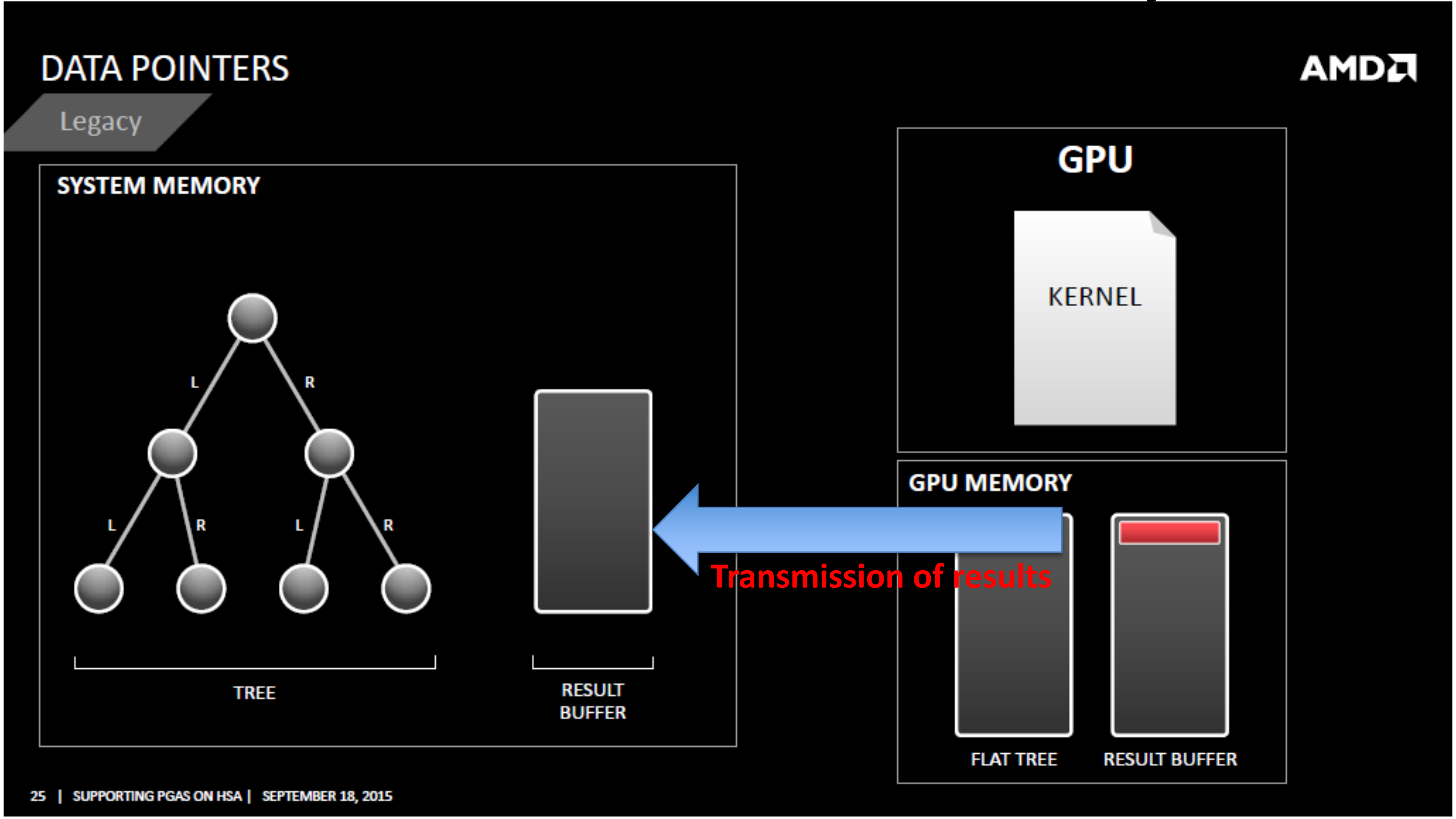
Legacy

23 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

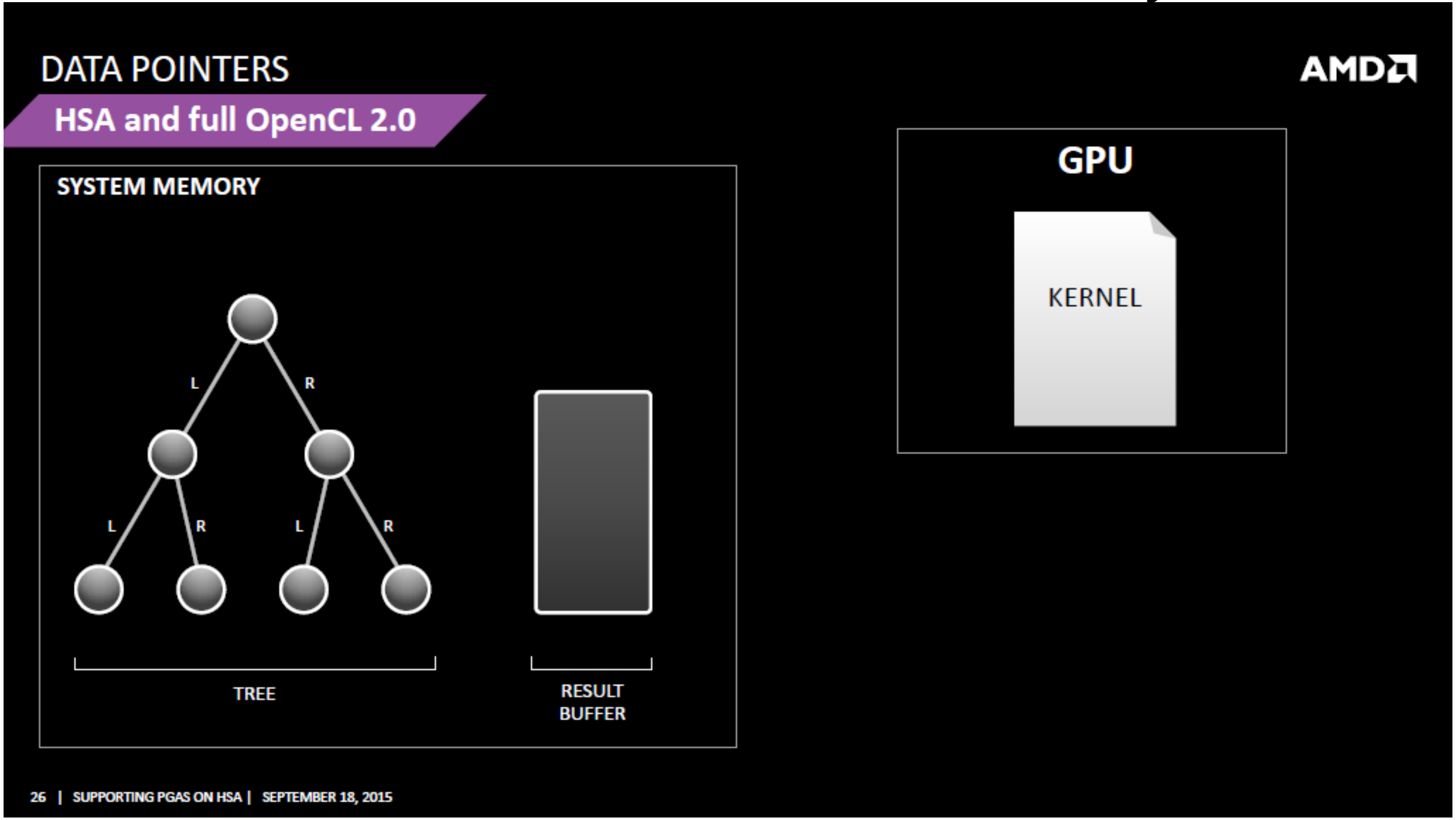
Unified coherent memory



Unified coherent memory



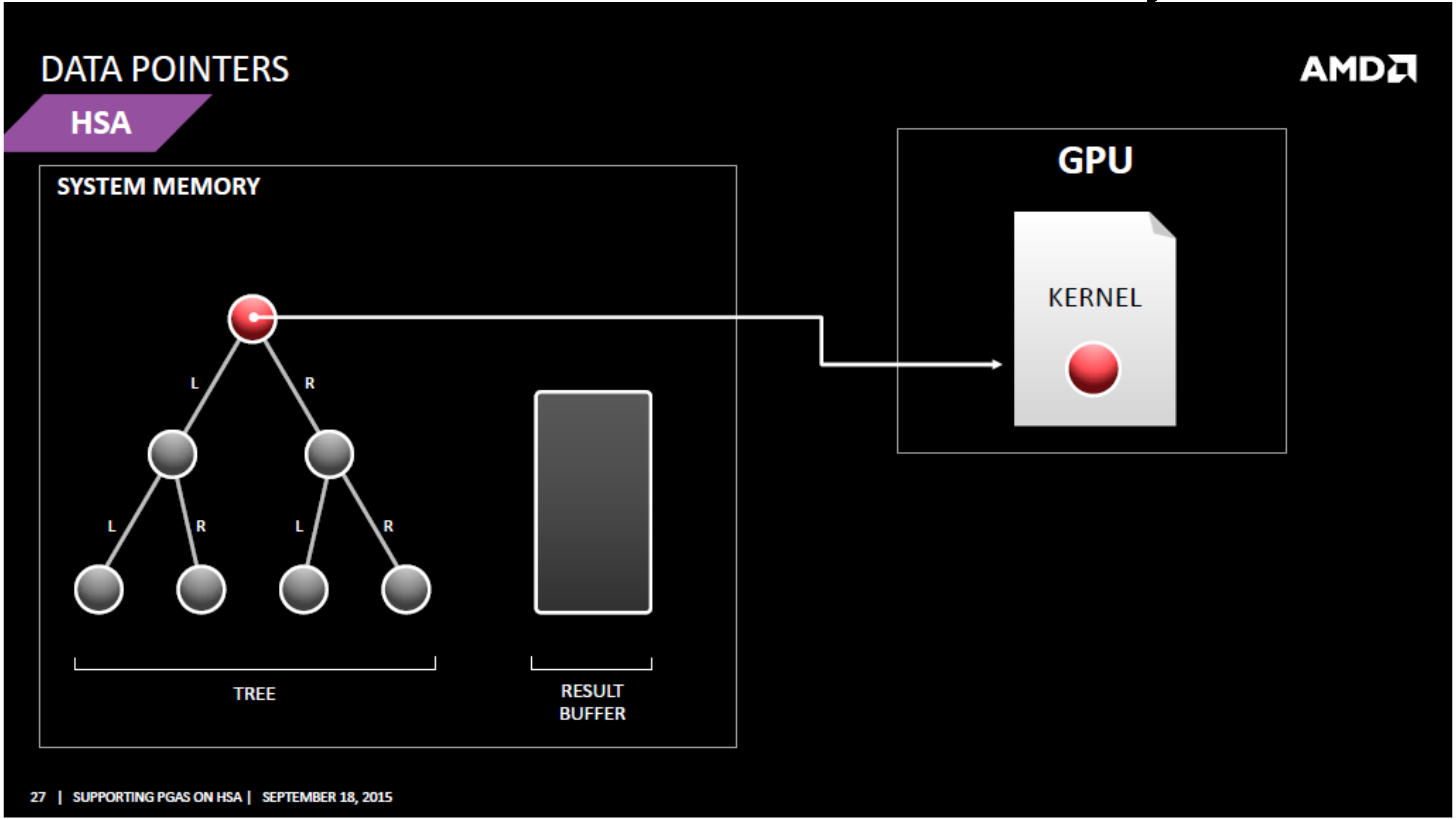
Unified coherent memory



DATA POINTERS

HSA and full OpenCL 2.0

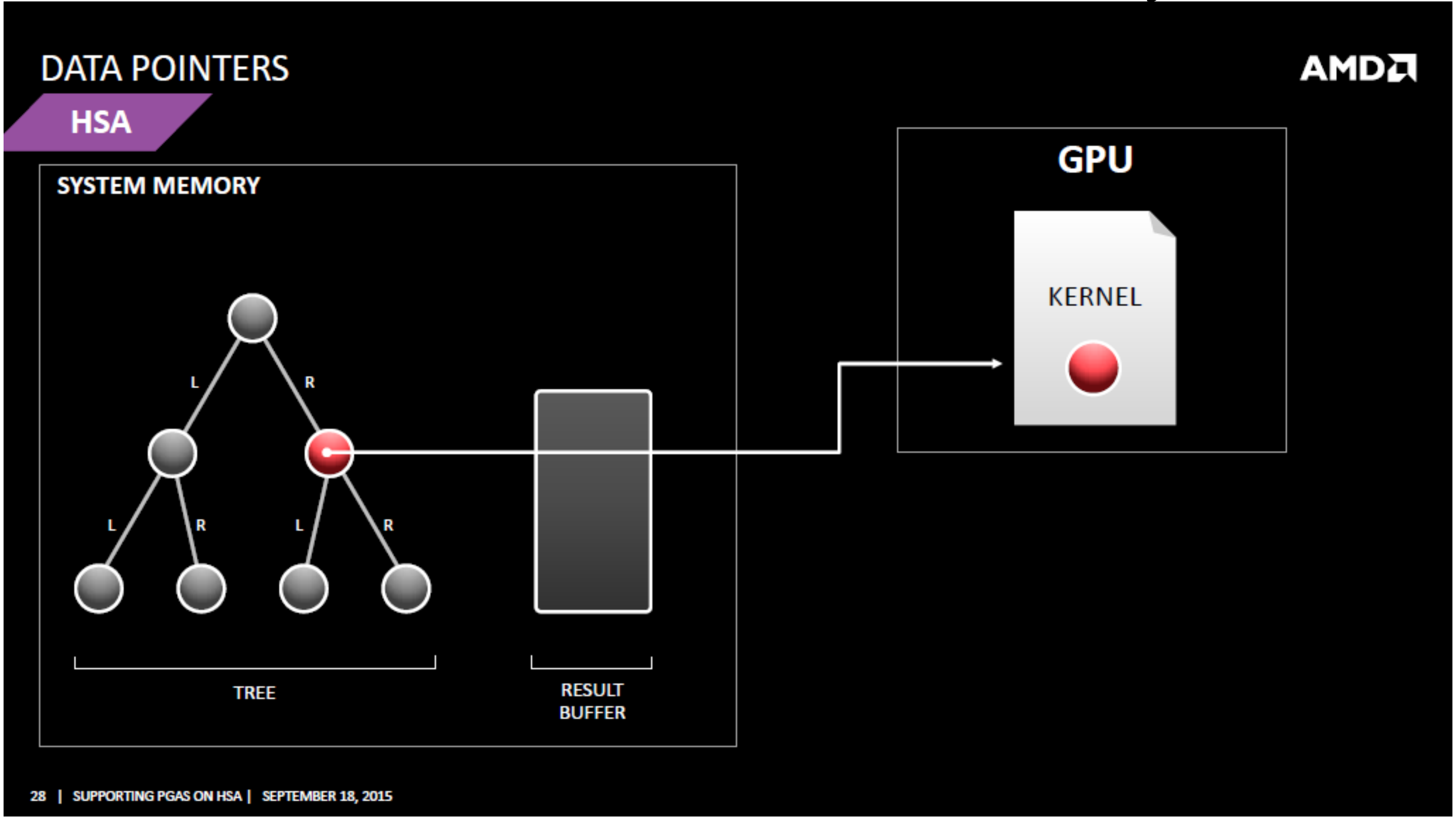
Unified coherent memory



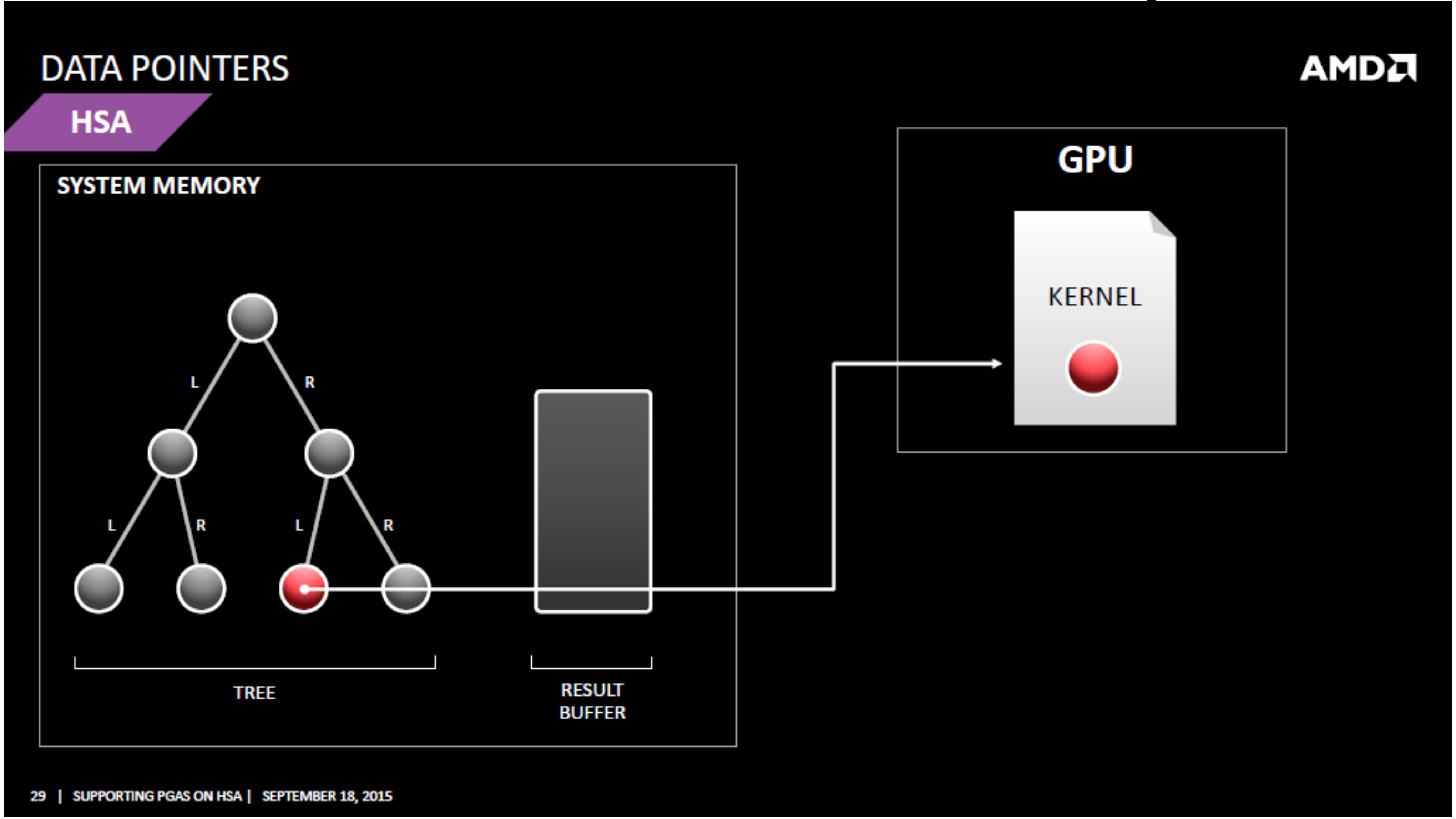
DATA POINTERS

HSA

Unified coherent memory



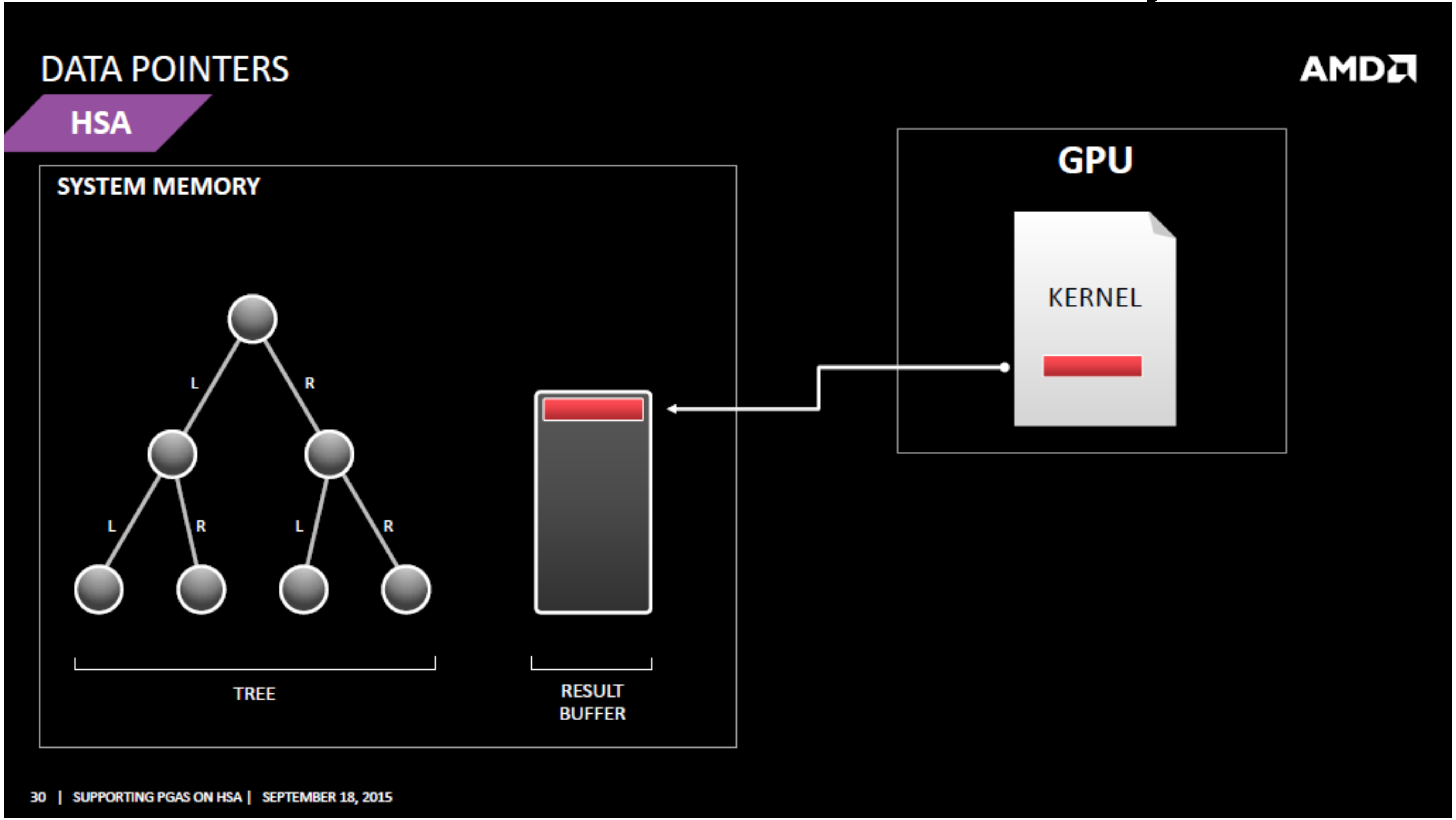
Unified coherent memory


DATA POINTERS

HSA

29 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

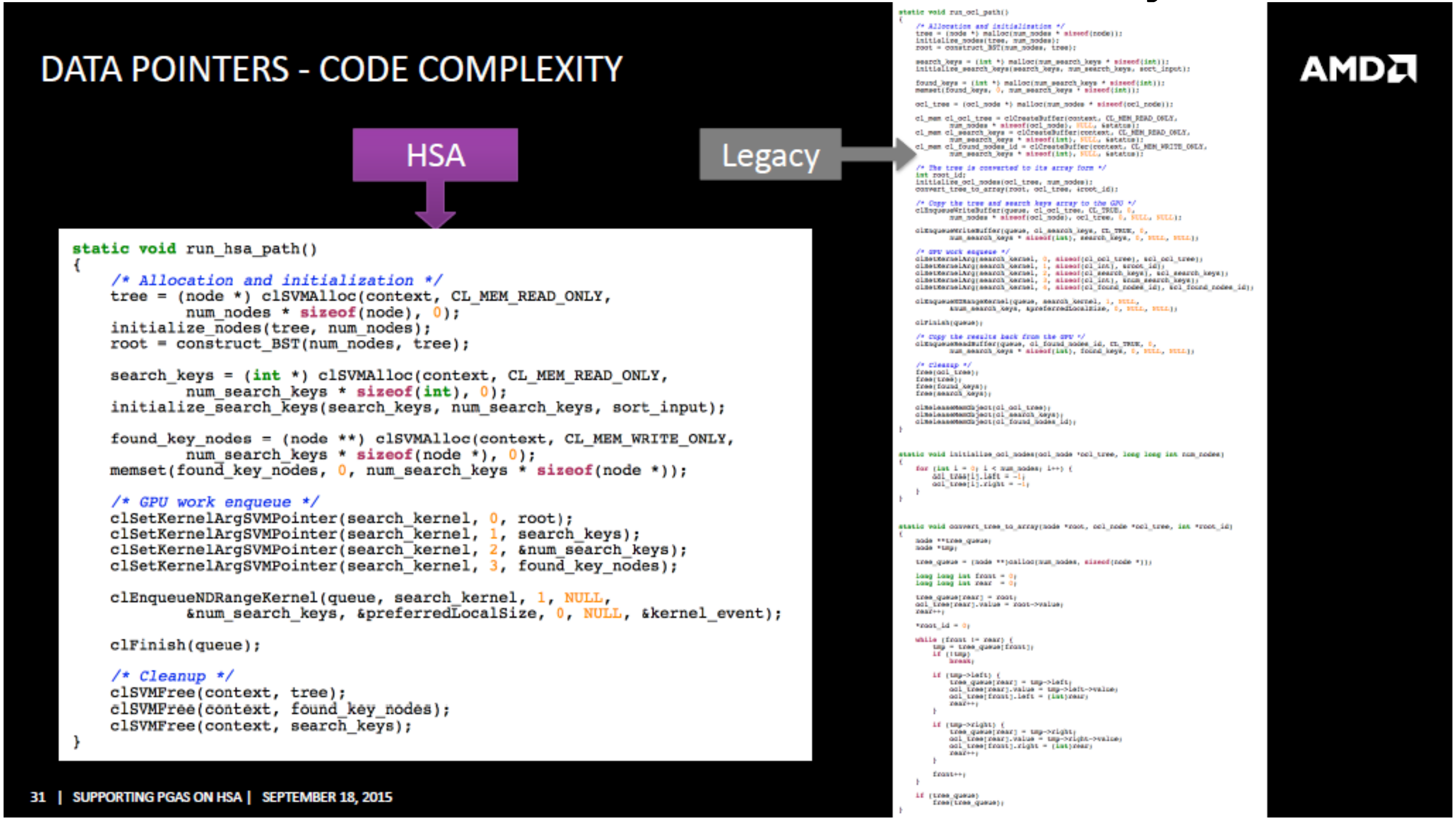
Unified coherent memory


DATA POINTERS

HSA

30 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

Unified coherent memory



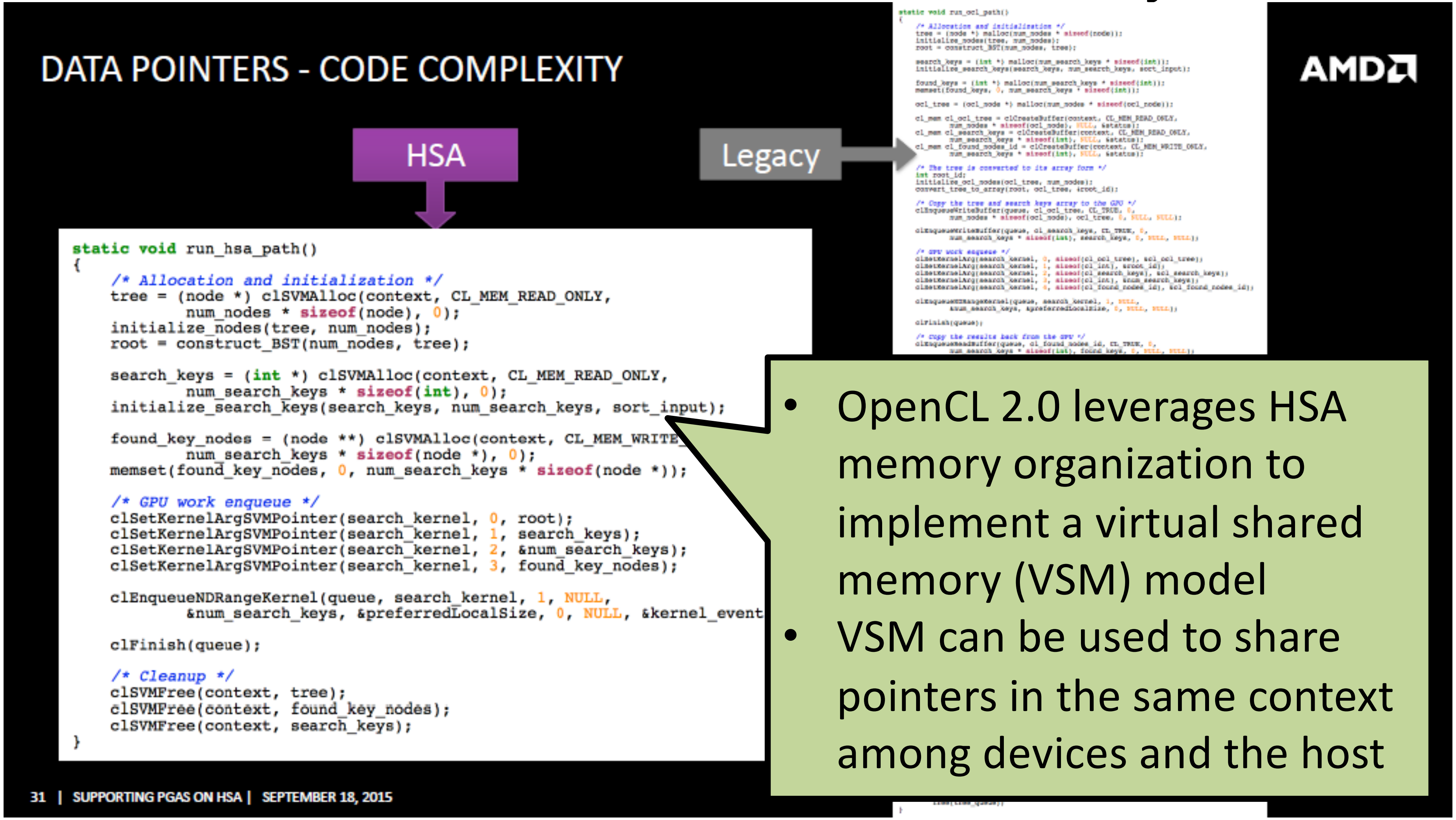
DATA POINTERS - CODE COMPLEXITY

```
static void run_hsa_path()
{
    /* Allocation and initialization */
    tree = (node *) clSVMAlloc(context, CL_MEM_READ_ONLY,
                num_nodes * sizeof(node), 0);
    initialize_nodes(tree, num_nodes);
    root = construct_BST(num_nodes, tree);

    search_keys = (int *) clSVMAlloc(context, CL_MEM_READ_ONLY,
                num_search_keys * sizeof(int), 0);
    initialize_search_keys(search_keys, num_search_keys, sort_input);

    found_key_nodes = (node **) clSVMAlloc(context, CL_MEM_WRITE_ONLY,
                num_search_keys * sizeof(node *), 0);
    memset(found_key_nodes, 0, num_search_keys * sizeof(node *));

    /* GPU work enqueue */
    clSetKernelArgSVMPointer(search_kernel, 0, root);
    clSetKernelArgSVMPointer(search_kernel, 1, search_keys);
    clSetKernelArgSVMPointer(search_kernel, 2, &num_search_keys);
    clSetKernelArgSVMPointer(search_kernel, 3, found_key_nodes);

    clEnqueueNDRangeKernel(queue, search_kernel, 1, NULL,
                &num_search_keys, &preferredLocalSize, 0, NULL, &kernel_event);

    clFinish(queue);

    /* Cleanup */
    clSVMFree(context, tree);
    clSVMFree(context, found_key_nodes);
    clSVMFree(context, search_keys);
}
```

```
static void run_ocl_path()
{
    /* Allocation and initialization */
    tree = (node *) malloc(num_nodes * sizeof(node));
    initialize_nodes(tree, num_nodes);
    root = construct_BST(num_nodes, tree);

    search_keys = (int *) malloc(num_search_keys * sizeof(int));
    initialize_search_keys(search_keys, num_search_keys, sort_input);

    found_keys = (int *) malloc(num_search_keys * sizeof(int));
    memset(found_keys, 0, num_search_keys * sizeof(int));

    ocl_tree = (ocl_node *) malloc(num_nodes * sizeof(ocl_node));

    cl_mem cl_ocl_tree = clCreateBuffer(context, CL_MEM_READ_ONLY,
            num_nodes * sizeof(ocl_node), NULL, &status);
    cl_mem cl_search_keys = clCreateBuffer(context, CL_MEM_READ_ONLY,
            num_search_keys * sizeof(int), NULL, &status);
    cl_mem cl_found_nodes_id = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            num_search_keys * sizeof(int), NULL, &status);

    /* The tree is converted to its array form */
    int root_id;
    initialize_ocl_nodes(ocl_tree, num_nodes);
    convert_tree_to_array(root, ocl_tree, &root_id);

    /* Copy the tree and search keys array to the GPU */
    clEnqueueWriteBuffer(queue, cl_ocl_tree, CL_TRUE, 0,
            num_nodes * sizeof(ocl_node), ocl_tree, 0, NULL, NULL);

    clEnqueueWriteBuffer(queue, cl_search_keys, CL_TRUE, 0,
            num_search_keys * sizeof(int), search_keys, 0, NULL, NULL);

    /* GPU work enqueue */
    clSetKernelArg(search_kernel, 0, sizeof(cl_ocl_tree), &cl_ocl_tree);
    clSetKernelArg(search_kernel, 1, sizeof(cl_int), &root_id);
    clSetKernelArg(search_kernel, 2, sizeof(cl_search_keys), &cl_search_keys);
    clSetKernelArg(search_kernel, 3, sizeof(cl_int), &num_search_keys);
    clSetKernelArg(search_kernel, 4, sizeof(cl_found_nodes_id), &cl_found_nodes_id);

    clEnqueueNDRangeKernel(queue, search_kernel, 1, NULL,
            &num_search_keys, &preferredLocalSize, 0, NULL, NULL);

    clFinish(queue);

    /* copy the results back from the GPU */
    clEnqueueReadBuffer(queue, cl_found_nodes_id, CL_TRUE, 0,
            num_search_keys * sizeof(int), found_keys, 0, NULL, NULL);

    /* cleanup */
    free(ocl_tree);
    free(tree);
    free(found_keys);
    free(search_keys);

    clReleaseMemObject(cl_ocl_tree);
    clReleaseMemObject(cl_search_keys);
    clReleaseMemObject(cl_found_nodes_id);
}


static void initialize_ocl_nodes(ocl_node *ocl_tree, long long int num_nodes)
{
    for (int i = 0; i < num_nodes; i++) {
        ocl_tree[i].left = -1;
        ocl_tree[i].right = -1;
    }
}


static void convert_tree_to_array(node *root, ocl_node *ocl_tree, int *root_id)
{
    node **tree_queue;
    node *tmp;

    tree_queue = (node **)calloc(num_nodes, sizeof(node *));

    long long int front = 0;
    long long int rear  = 0;

    tree_queue[rear] = root;
    ocl_tree[rear].value = root->value;
    rear++;

    *root_id = 0;

    while (front != rear) {
        tmp = tree_queue[front];
        if (!tmp)
            break;

        if (tmp->left) {
            tree_queue[rear] = tmp->left;
            ocl_tree[rear].value = tmp->left->value;
            ocl_tree[front].left = (int)rear;
            rear++;
        }

        if (tmp->right) {
            tree_queue[rear] = tmp->right;
            ocl_tree[rear].value = tmp->right->value;
            ocl_tree[front].right = (int)rear;
            rear++;
        }

        front++;
    }

    if (tree_queue)
        free(tree_queue);
}
```

31 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

Unified coherent memory



DATA POINTERS - CODE COMPLEXITY

AMD

```
static void run_hsa_path()
{
    /* Allocation and initialization */
    tree = (node *) clSVMAlloc(context, CL_MEM_READ_ONLY,
            num_nodes * sizeof(node), 0);
    initialize_nodes(tree, num_nodes);
    root = construct_BST(num_nodes, tree);

    search_keys = (int *) clSVMAlloc(context, CL_MEM_READ_ONLY,
            num_search_keys * sizeof(int), 0);
    initialize_search_keys(search_keys, num_search_keys, sort_input);

    found_key_nodes = (node **) clSVMAlloc(context, CL_MEM_WRITE_
            num_search_keys * sizeof(node *), 0);
    memset(found_key_nodes, 0, num_search_keys * sizeof(node *));

    /* GPU work enqueue */
    clSetKernelArgSVMPointer(search_kernel, 0, root);
    clSetKernelArgSVMPointer(search_kernel, 1, search_keys);
    clSetKernelArgSVMPointer(search_kernel, 2, &num_search_keys);
    clSetKernelArgSVMPointer(search_kernel, 3, found_key_nodes);

    clEnqueueNDRangeKernel(queue, search_kernel, 1, NULL,
            &num_search_keys, &preferredLocalSize, 0, NULL, &kernel_event

    clFinish(queue);

    /* Cleanup */
    clSVMFree(context, tree);
    clSVMFree(context, found_key_nodes);
    clSVMFree(context, search_keys);
}
```

```
static void run_ocl_path()
{
    /* Allocation and initialization */
    tree = (node *) malloc(num_nodes * sizeof(node));
    initialize_nodes(tree, num_nodes);
    root = construct_BST(num_nodes, tree);

    search_keys = (int *) malloc(num_search_keys * sizeof(int));
    initialize_search_keys(search_keys, num_search_keys, sort_input);

    found_keys = (int *) malloc(num_search_keys * sizeof(int));
    memset(found_keys, 0, num_search_keys * sizeof(int));

    ocl_tree = (ocl_node *) malloc(num_nodes * sizeof(ocl_node));

    cl_mem cl_ocl_tree = clCreateBuffer(context, CL_MEM_READ_ONLY,
            num_nodes * sizeof(ocl_node), NULL, &status);
    cl_mem cl_search_keys = clCreateBuffer(context, CL_MEM_READ_ONLY,
            num_search_keys * sizeof(int), NULL, &status);
    cl_mem cl_found_nodes_id = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            num_search_keys * sizeof(int), NULL, &status);

    /* The tree is converted to its array form */
    int root_id;
    initialize_ocl_nodes(ocl_tree, num_nodes);
    convert_tree_to_array(root, ocl_tree, &root_id);

    /* Copy the tree and search keys array to the GPU */
    clEnqueueWriteBuffer(queue, cl_ocl_tree, CL_TRUE, 0,
            num_nodes * sizeof(ocl_node), ocl_tree, 0, NULL, NULL);

    clEnqueueWriteBuffer(queue, cl_search_keys, CL_TRUE, 0,
            num_search_keys * sizeof(int), search_keys, 0, NULL, NULL);

    /* GPU work enqueue */
    clSetKernelArg(search_kernel, 0, sizeof(cl_ocl_tree), &cl_ocl_tree);
    clSetKernelArg(search_kernel, 1, sizeof(cl_int), &root_id);
    clSetKernelArg(search_kernel, 2, sizeof(cl_search_keys), &cl_search_keys);
    clSetKernelArg(search_kernel, 3, sizeof(cl_int), &num_search_keys);
    clSetKernelArg(search_kernel, 4, sizeof(cl_found_nodes_id), &cl_found_nodes_id);

    clEnqueueNDRangeKernel(queue, search_kernel, 1, NULL,
            &num_search_keys, &preferredLocalSize, 0, NULL, NULL);

    clFinish(queue);

    /* Copy the results back from the GPU */
    clEnqueueReadBuffer(queue, cl_found_nodes_id, CL_TRUE, 0,
            num_search_keys * sizeof(int), found_keys, 0, NULL, NULL);
```

31 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

- OpenCL 2.0 leverages HSA memory organization to implement a virtual shared memory (VSM) model
- VSM can be used to share pointers in the same context among devices and the host

Key features of HSA

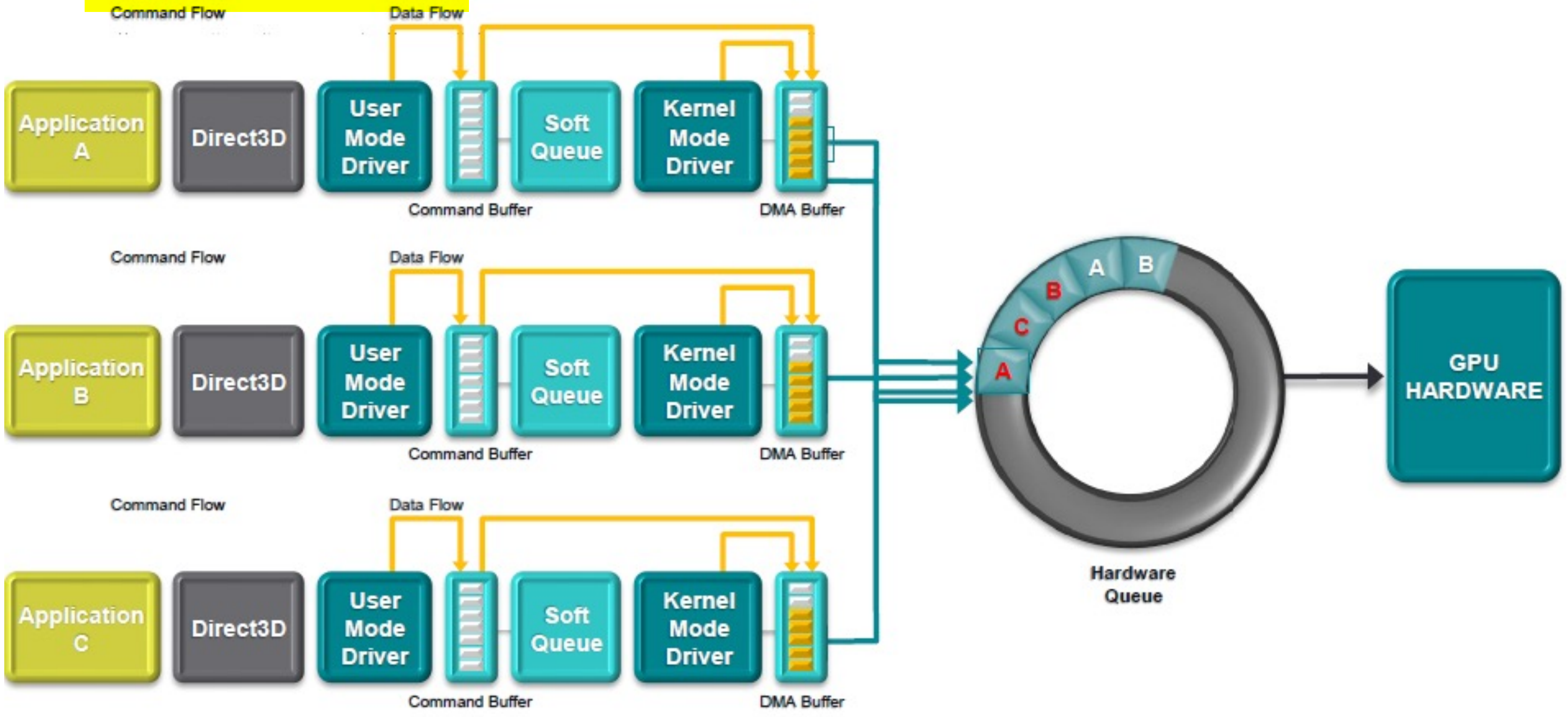
- hUMA – Heterogeneous Unified Memory Architecture
- hQ – Heterogeneous Queuing
- HSAIL – HSA Intermediate Language

hQ: heterogeneous queuing

- Task queuing runtimes
  - Popular pattern for task and data parallel programming on Symmetric Multiprocessor (SMP) systems
  - Characterized by:
    - A work queue per core
    - Runtime library that divides large loops into tasks and distributes to queues
    - A work stealing scheduler that keeps system balanced
- HSA is designed to extend this pattern to run on heterogeneous systems

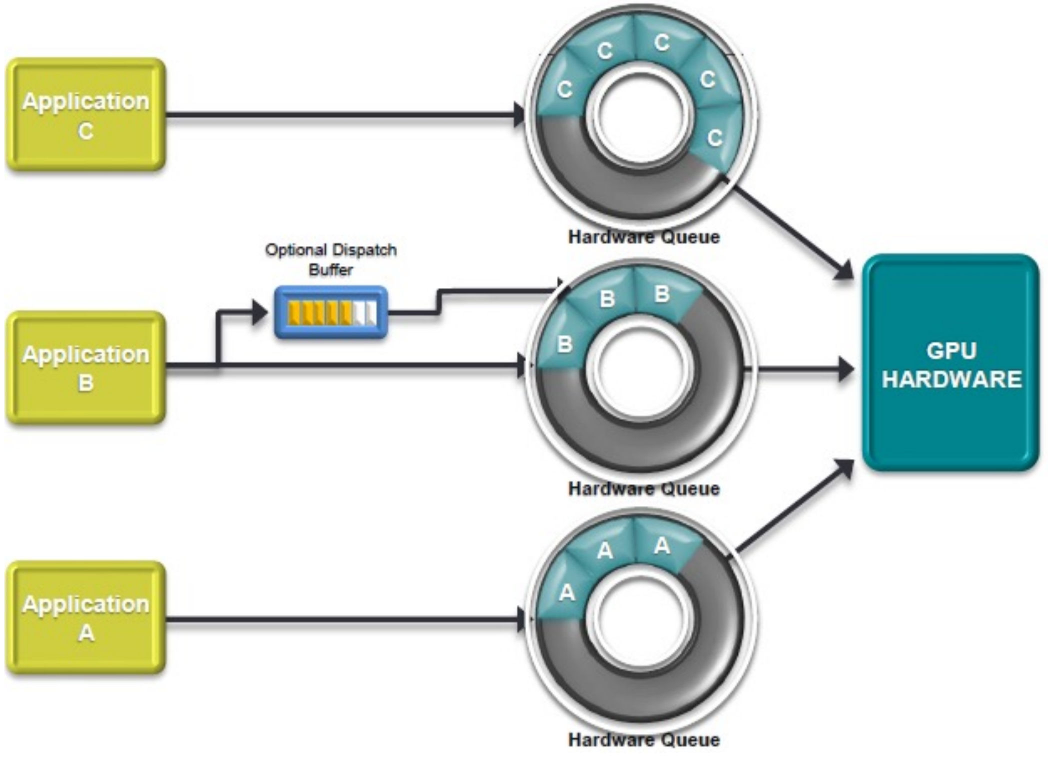
hQ: heterogeneous queuing

- How compute dispatch operates today in the driver model



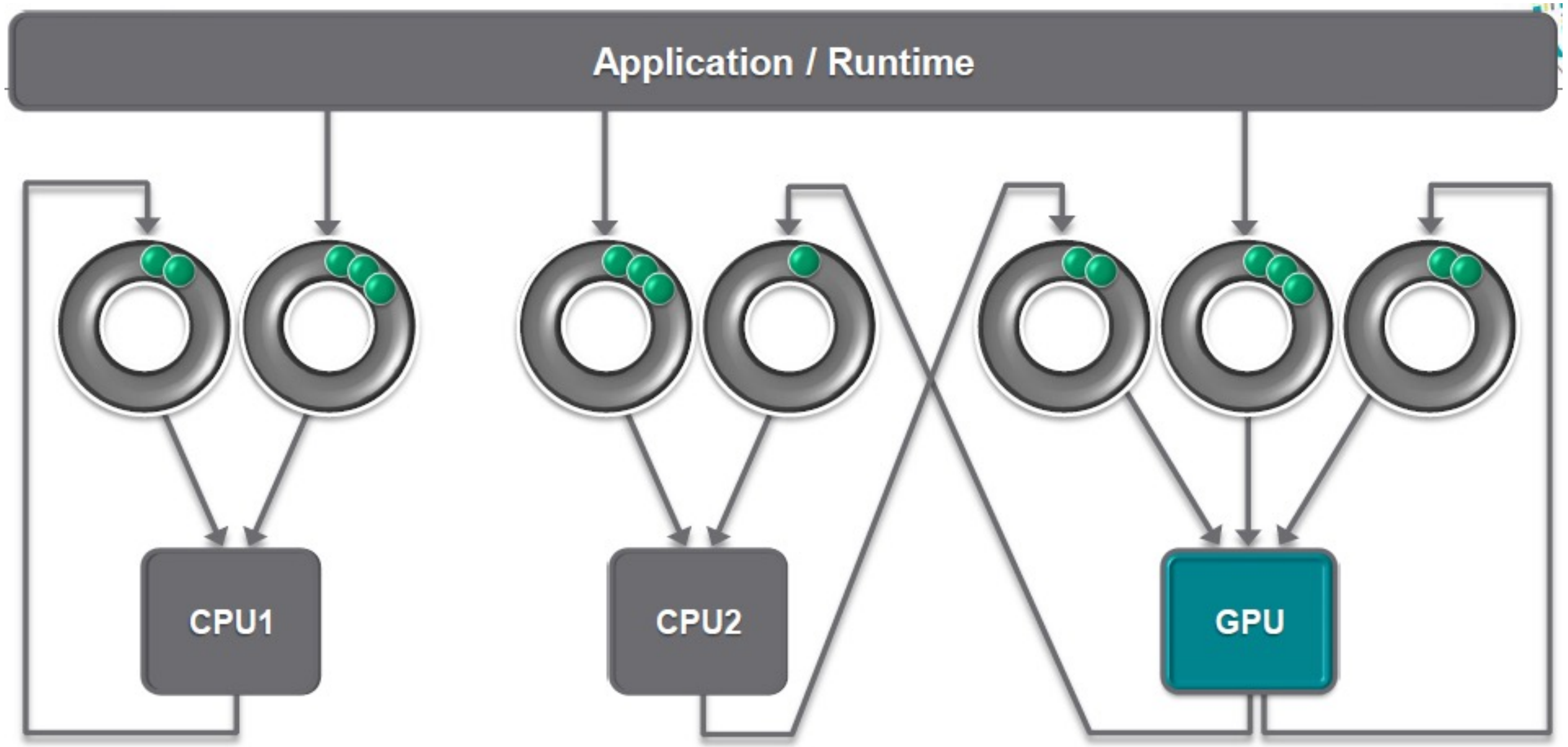
hQ: heterogeneous queuing

- How compute dispatch improves under HSA
  - Application codes to the hardware
  - User mode queuing
  - Hardware scheduling
  - Low dispatch times
  - No Soft Queues
  - No User Mode Drivers
  - No Kernel Mode Transitions
  - No Overhead!



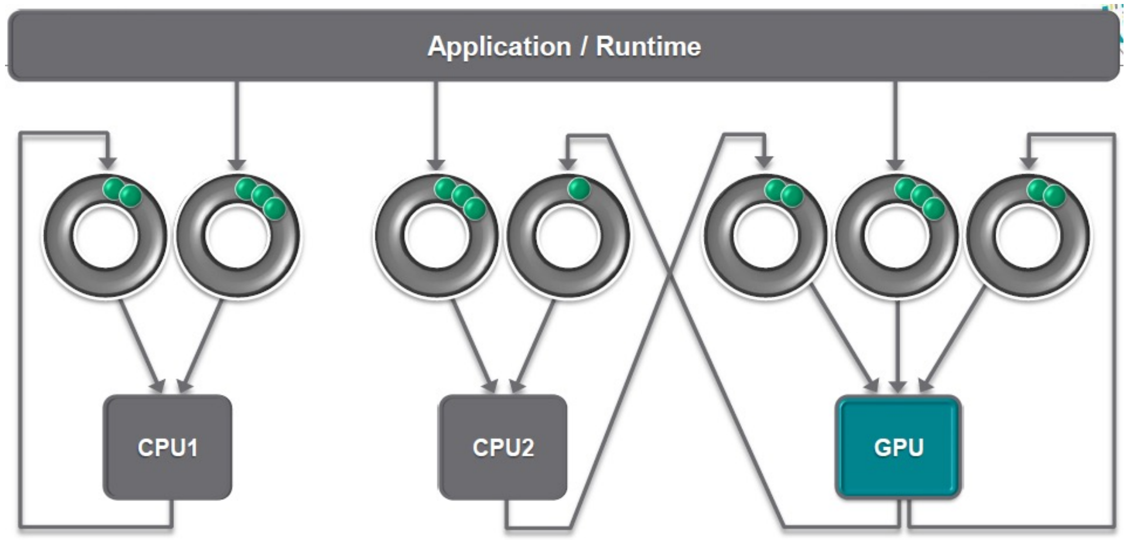
hQ: heterogeneous queuing

- AQL (Architected Queueing Layer) enables any agent to enqueue tasks



hQ: heterogeneous queuing

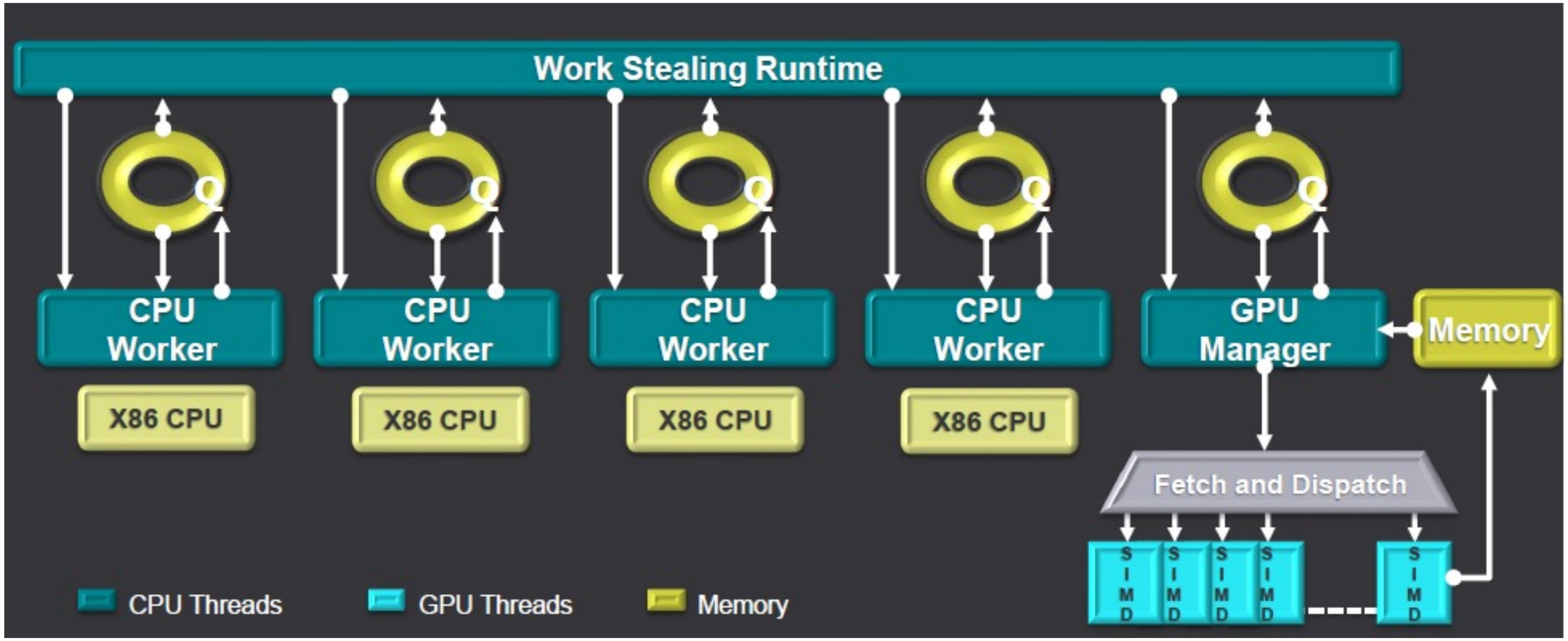
- AQL (Architected Queueing Layer) enables any agent to enqueue tasks
  - Single compute dispatch path for all hardware
  - No driver translation, direct access to hardware
  - Standard across vendors



- All agents can enqueue
  - Allowed also self-enqueuing
- Requires coherency and shared virtual memory

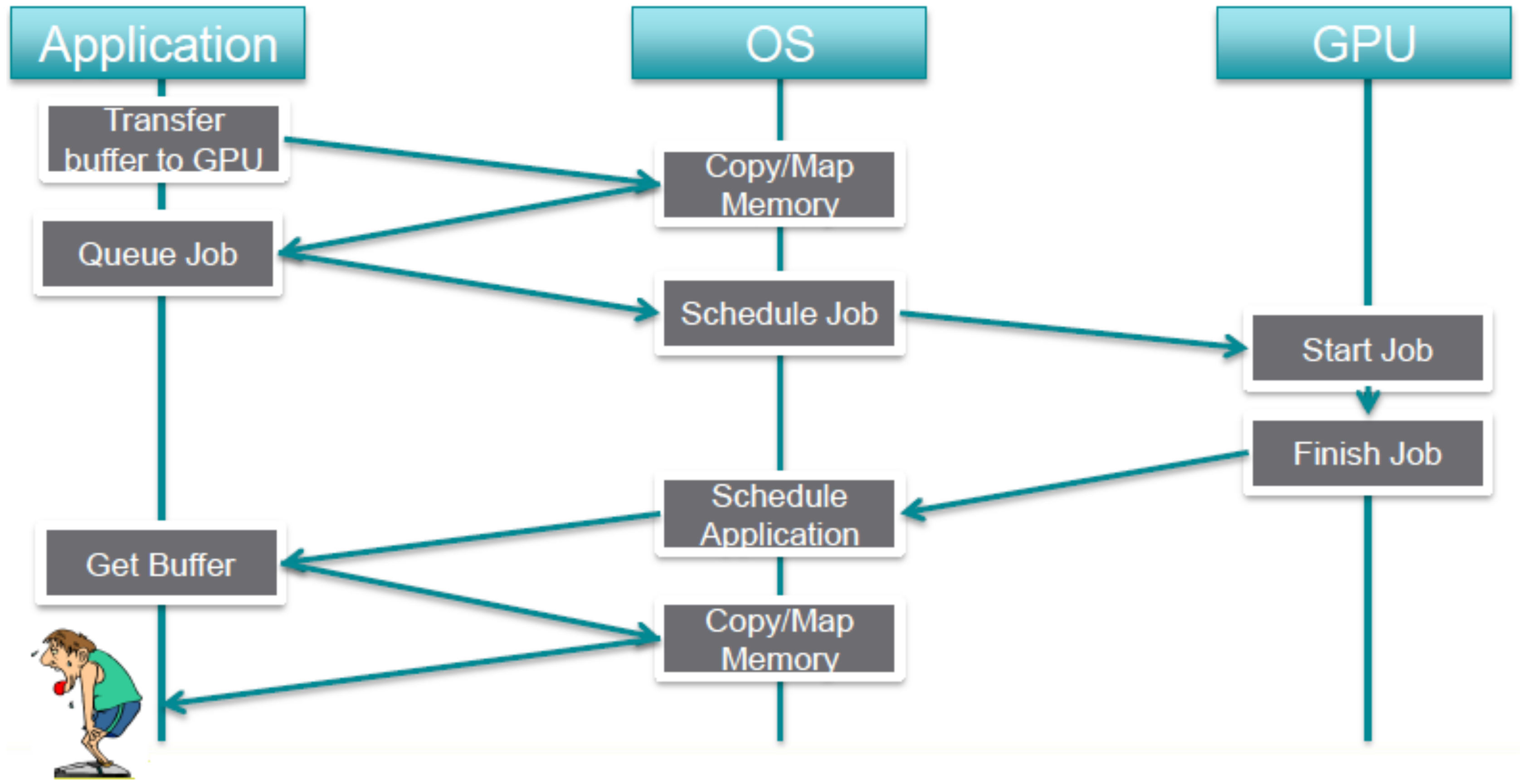
hQ: heterogeneous queuing

- A work stealing scheduler that keeps system balanced

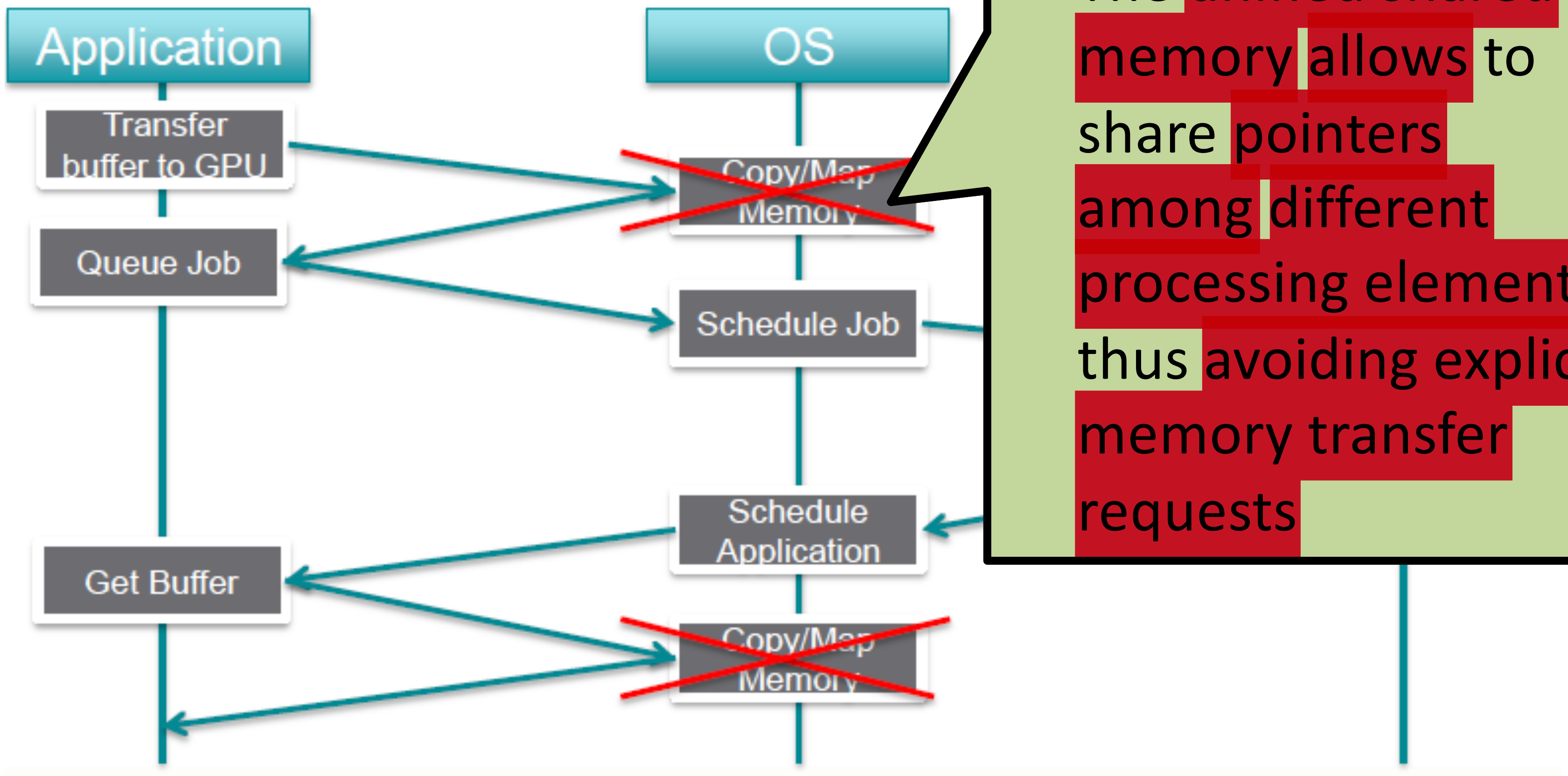


Advantages of the queuing model

- Today's picture:

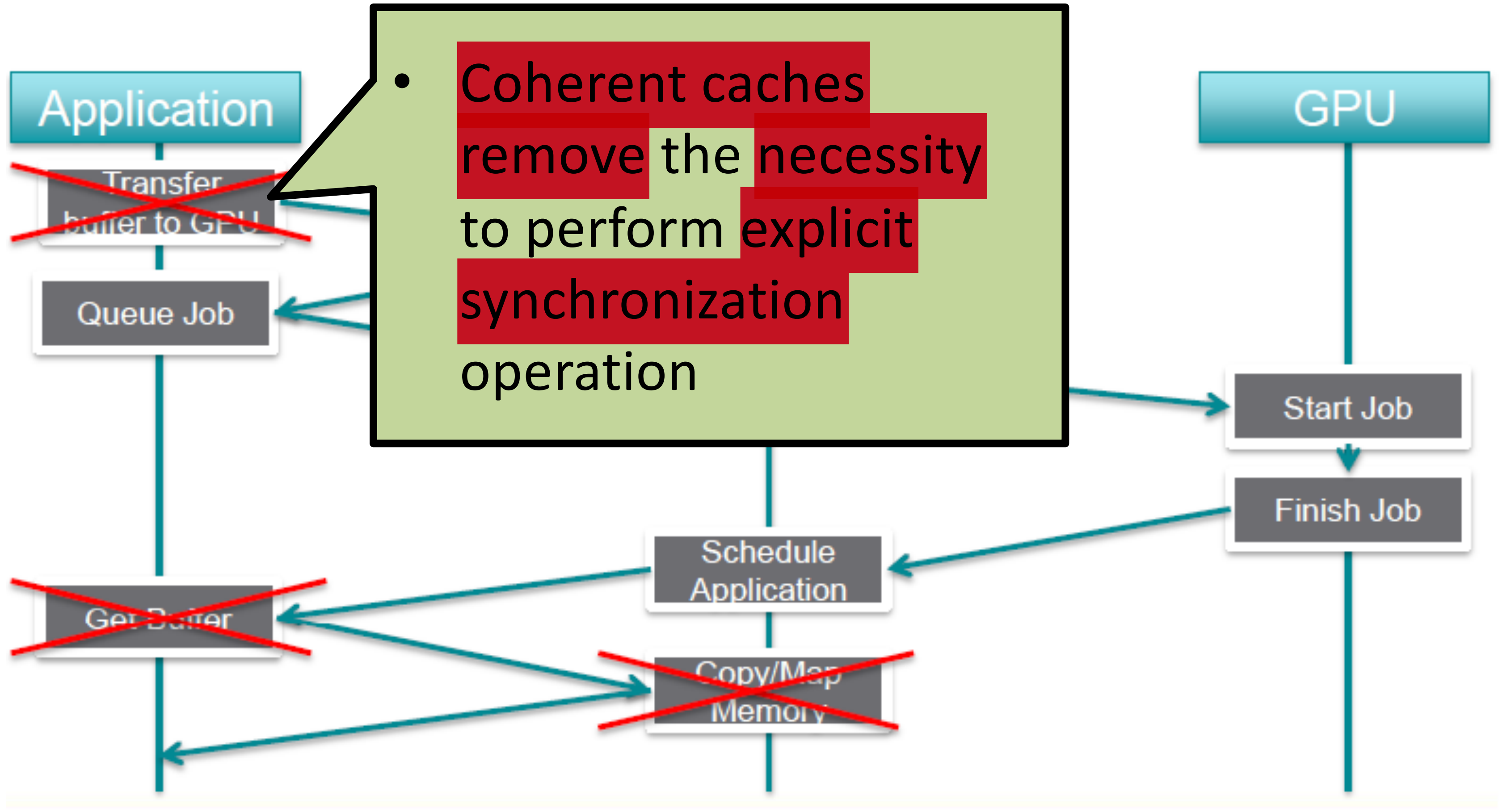


Advantages of the queuing model



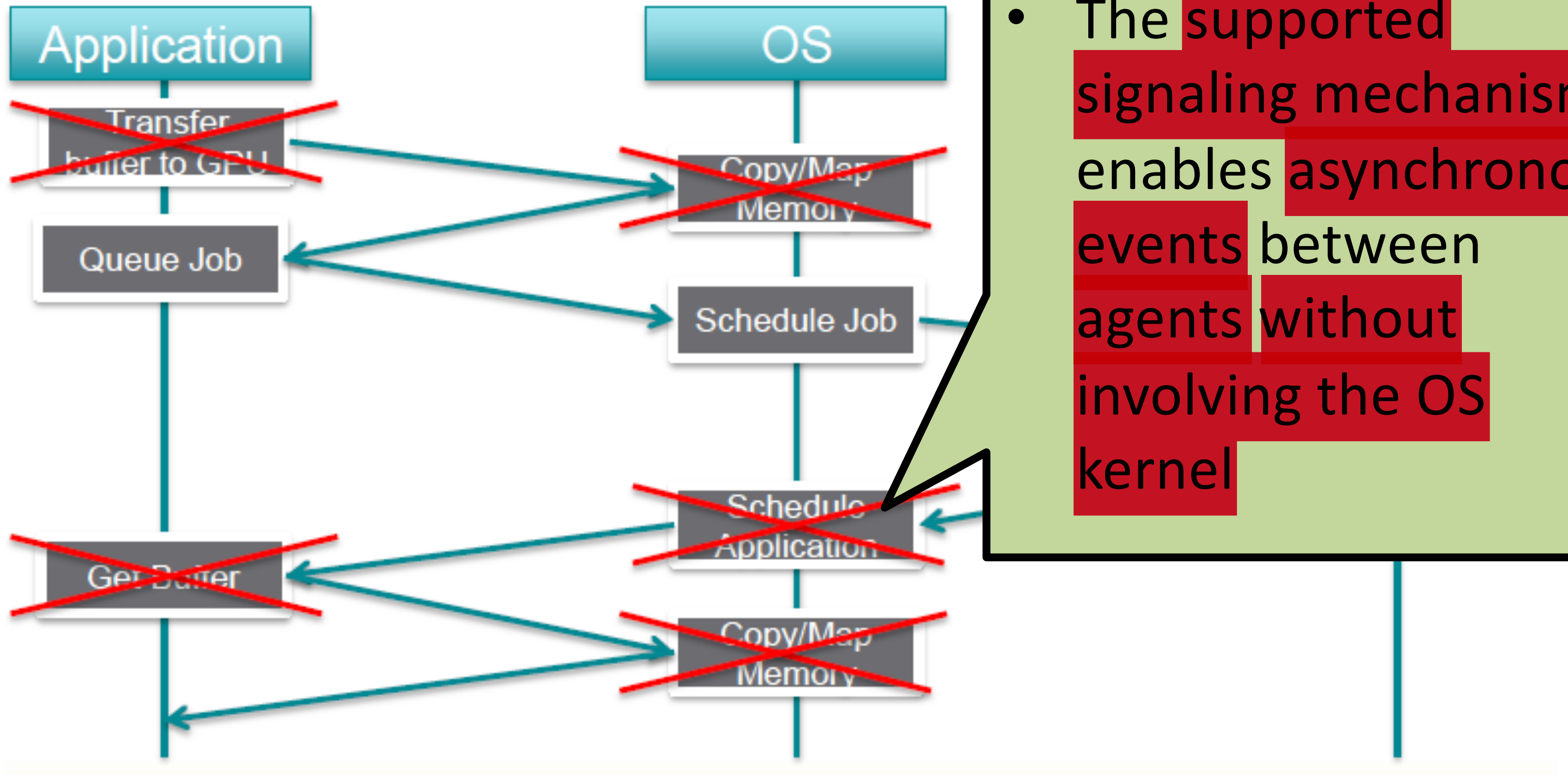
- The unified shared memory allows to share pointers among different processing elements thus avoiding explicit memory transfer requests

Advantages of the queuing model



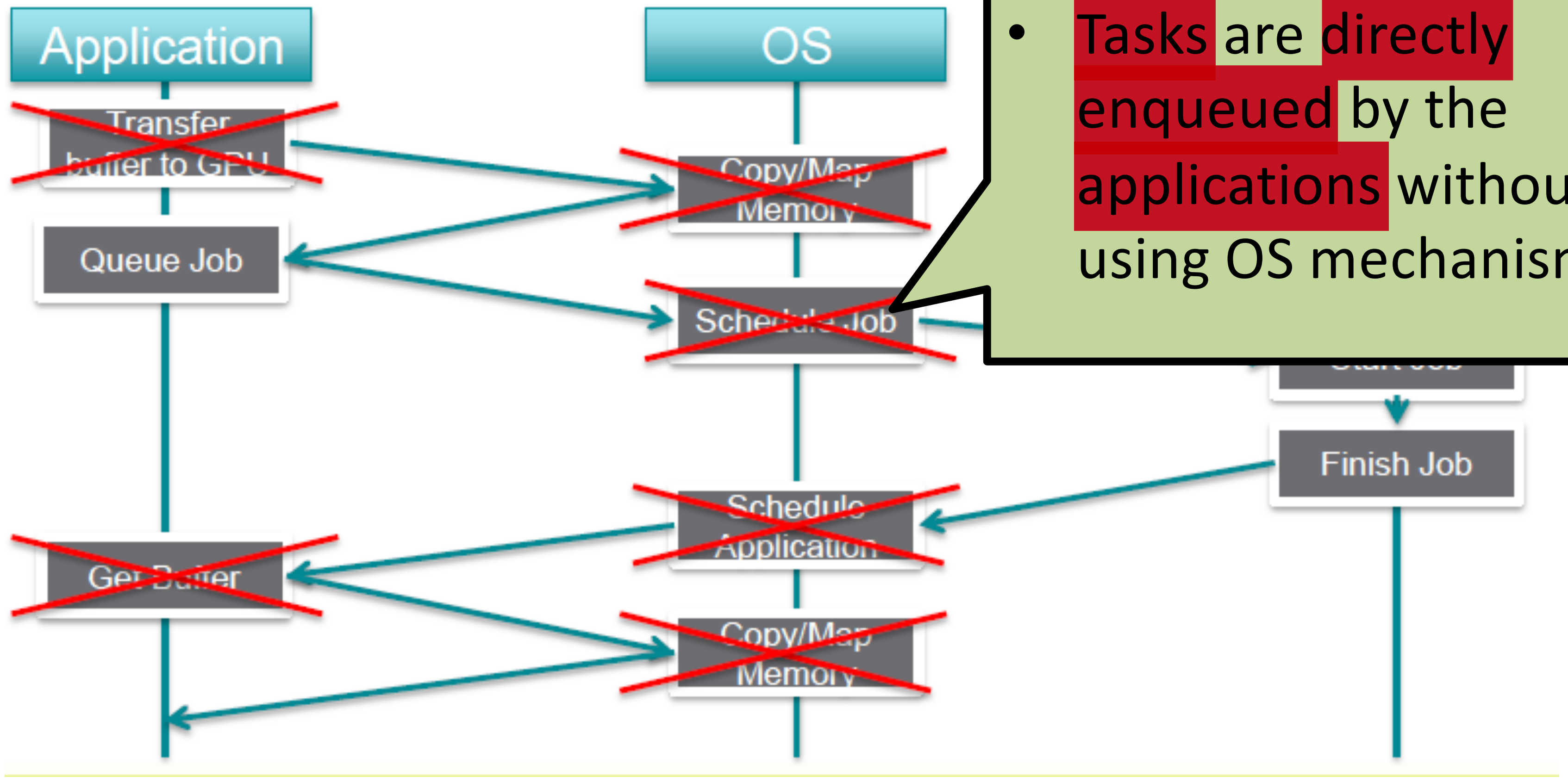
- Coherent caches remove the necessity to perform explicit synchronization operation

Advantages of the queuing model



- The supported signaling mechanism enables asynchronous events between agents without involving the OS kernel

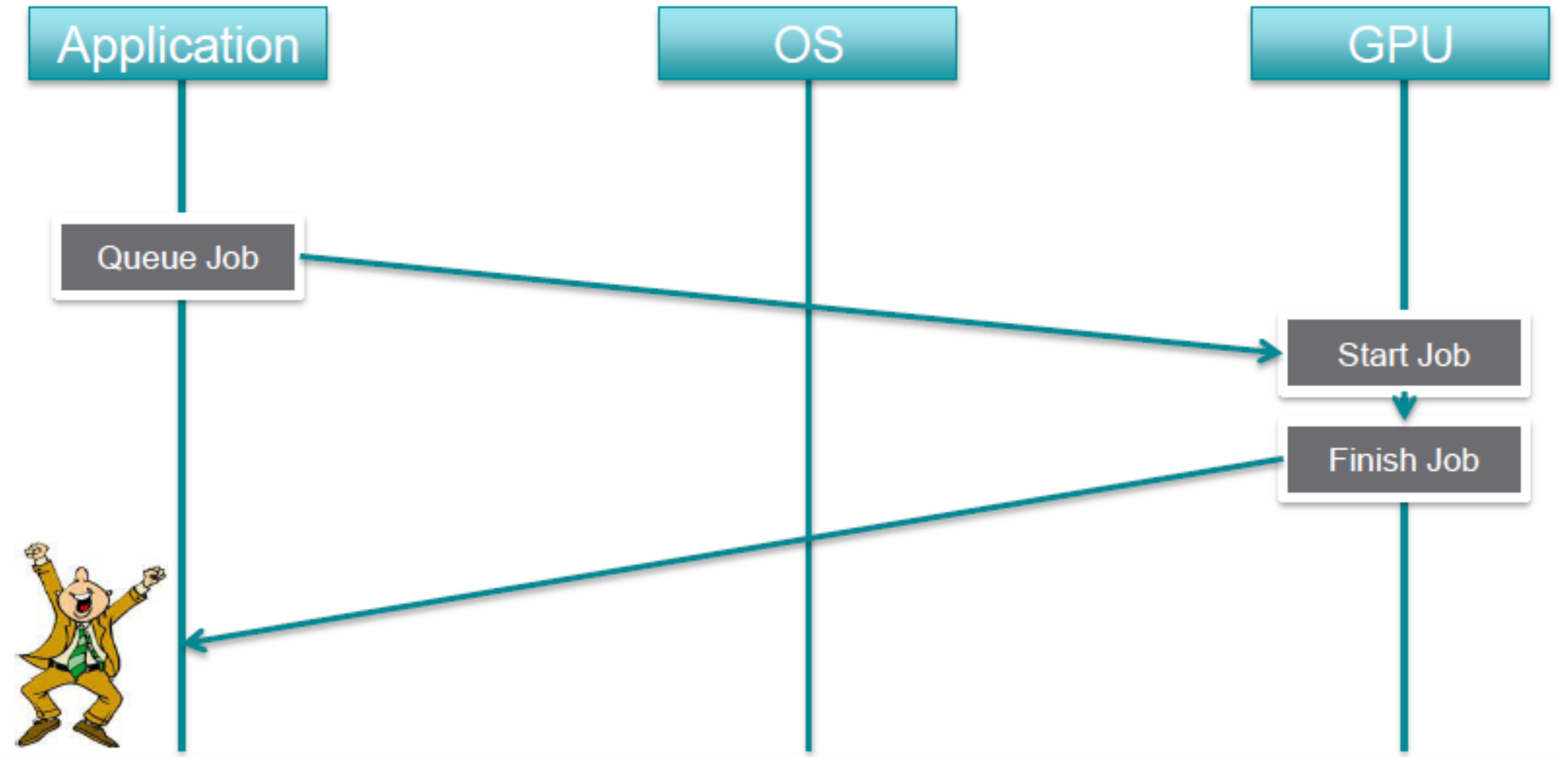
Advantages of the queuing model



- Tasks are directly enqueued by the applications without using OS mechanisms

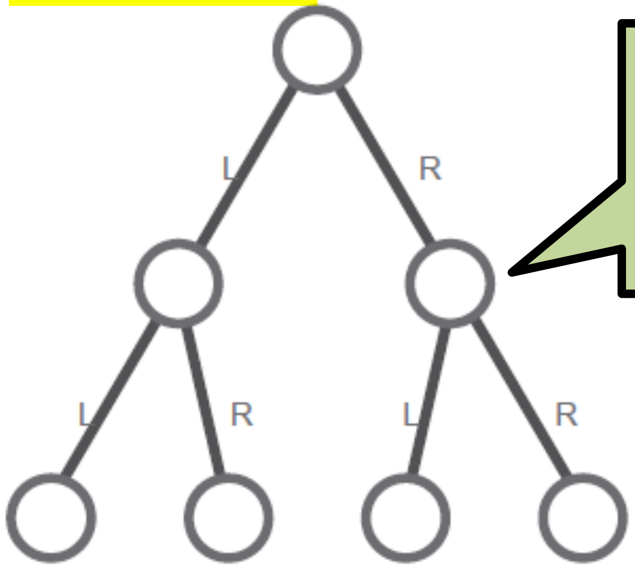
Advantages of the queuing model

- HSA picture:



Device side queuing

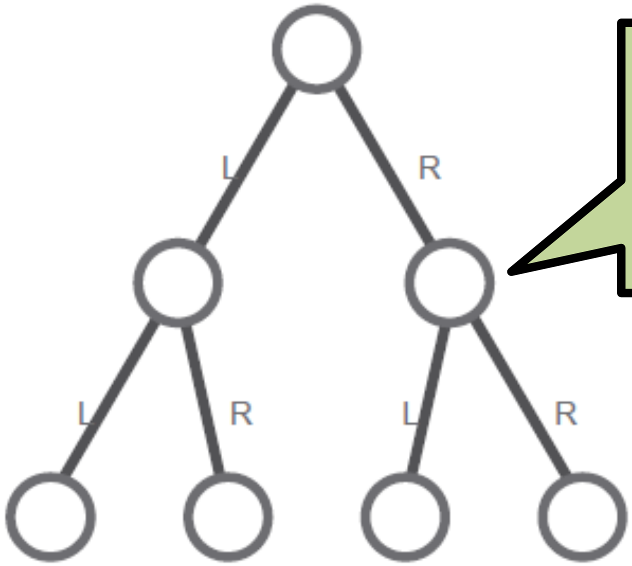
- Let's consider a tree traversal problem:
  - Every node in the tree is a job to be executed
  - We may not know at priory the size of the tree
  - Input parameters of a job may depend on parent execution



- Each node is a job
- Each job may generate some child jobs

Device side queuing

- State-of-the-art solution:
  - The job has to communicate to the host the new jobs (possibly transmitting input data)
  - The host queues the child jobs on the device

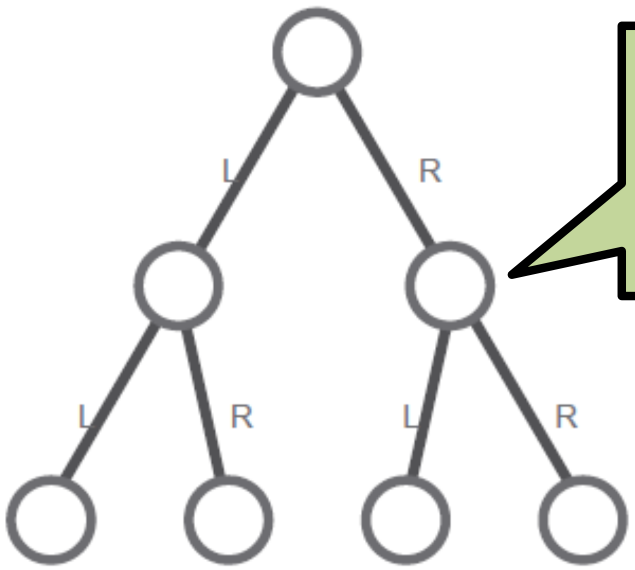


- Each node is a job
- Each job may generate some child jobs

Considerable memory traffic!

Device side queuing

- Device side queuing:
  - The job running on the device directly queues new jobs in the device/host queues



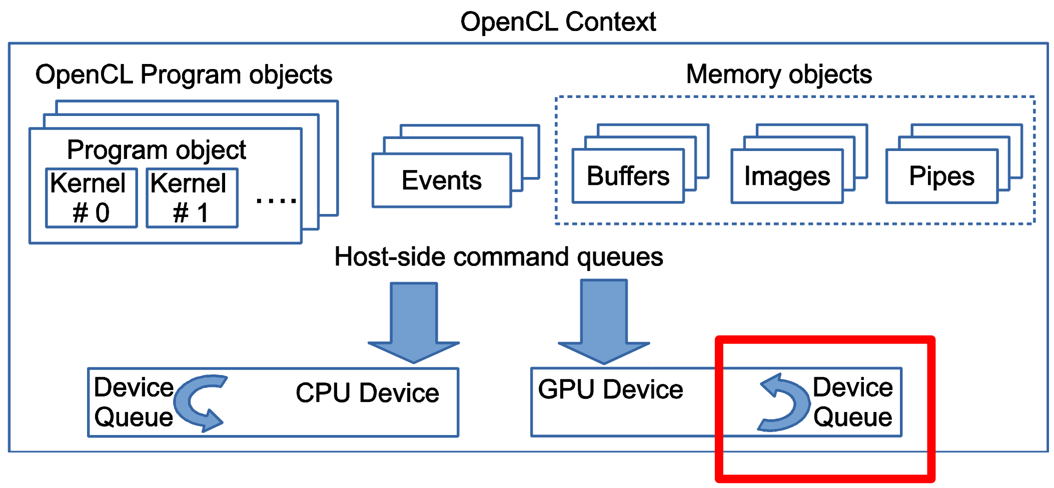
- Each node is a job
- Each job may generate some child jobs

Device side queuing

- Benefits of device side queuing:
  - Enable more natural expression of nested parallelism necessary for applications with irregular or data-driven loop structures(i.e., breadth first search)
  - Remove of synchronization and communication with the host to launch new threads (remove expensive data transfer)
  - The finer granularities of parallelism is exposed to scheduler and load balancer

Device side queuing

- OpenCL 2.0 supports device side queuing
  - Device-side command queues are out-of-order
  - Parent and child kernels execute asynchronously
  - Synchronization has to be explicitly managed by the programmer



Summary on the queuing model

- User mode queuing for low latency dispatch
  - Application dispatches directly
  - No OS or driver required in the dispatch path
- Architected Queuing Layer
  - Single compute dispatch path for all hardware
  - No driver translation, direct to hardware
- Allows for dispatch to queue from any agent
  - CPU or GPU
- GPU self-enqueue enables lots of solutions
  - Recursion
  - Tree traversal
  - Wavefront reforming

Other necessary HW mechanisms

- Task preemption and context switching have to be supported by all computing resources (also GPUs)

Key features of HSA

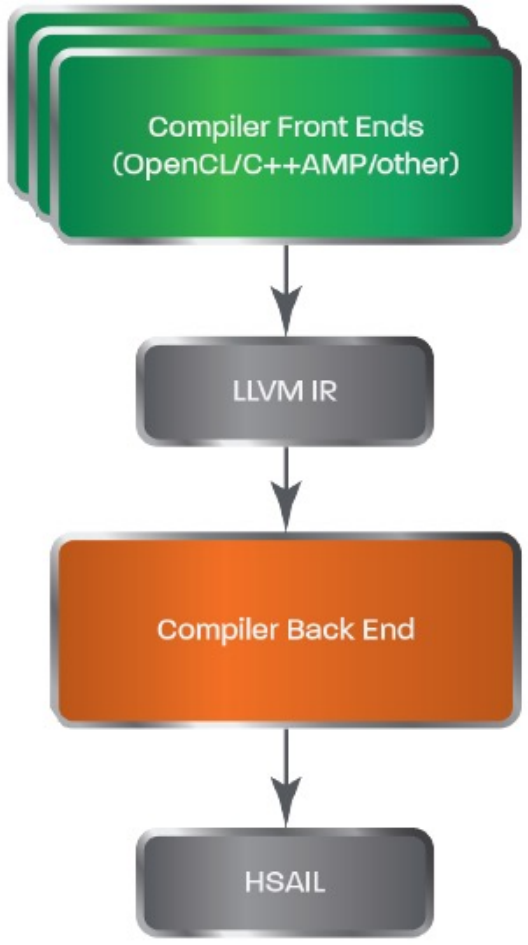
- hUMA – Heterogeneous Unified Memory Architecture
- hQ – Heterogeneous Queuing
- HSAIL – HSA Intermediate Language

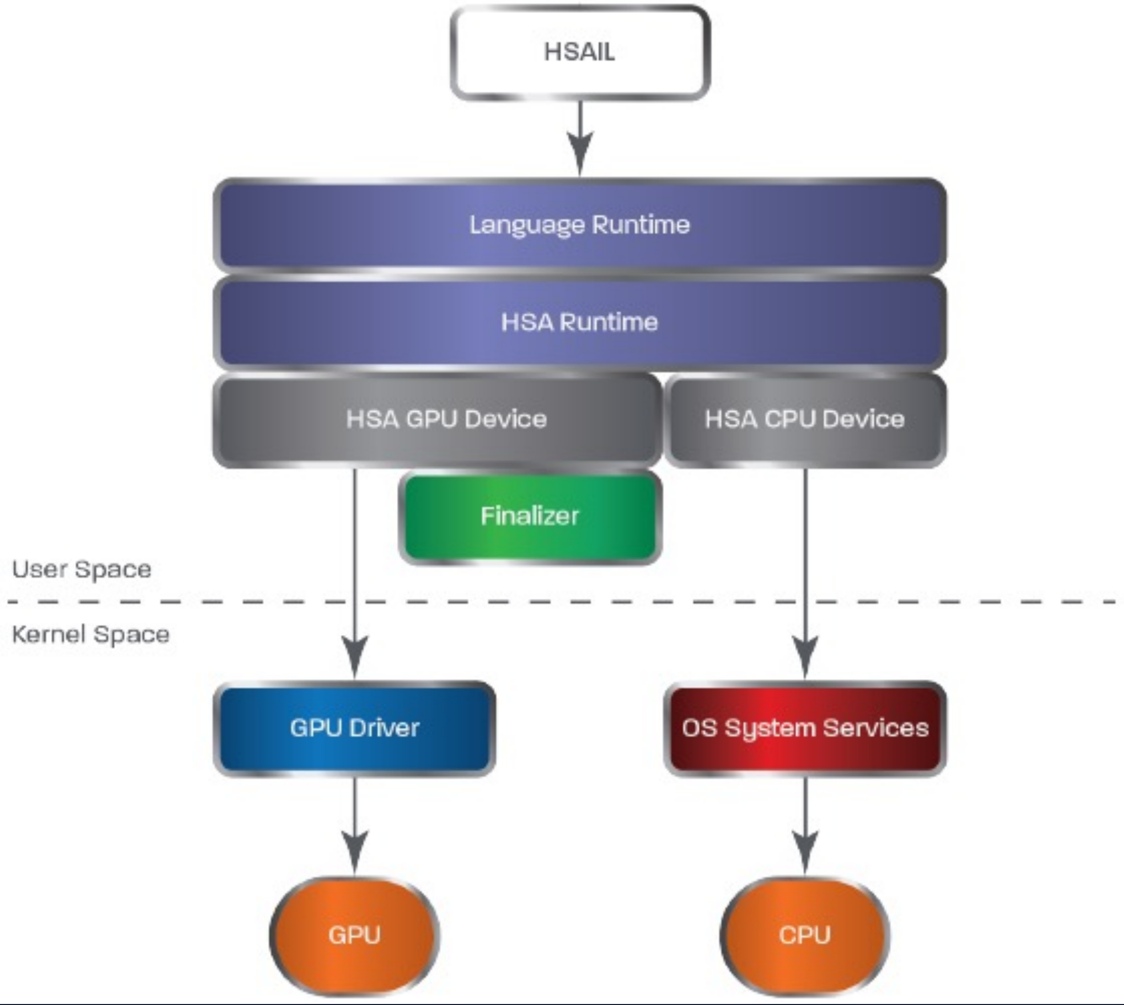
HSA intermediate layer (HSAIL)

- A portable "virtual ISA" for vendor-independent compilation and distribution
  - Like Java bytecodes for GPUs
- Low-level IR, close to machine ISA level
  - Most optimizations (including register allocation) performed before HSAIL
- Generated by a high-level compiler (LLVM, gcc, Java VM, etc.)
  - Application binaries may ship with embedded HSAIL
- Compiled down to target ISA by a vendor-specific "finalizer"
  - Finalizer may execute at run time, install time, or build time

HSA intermediate layer (HSAIL)

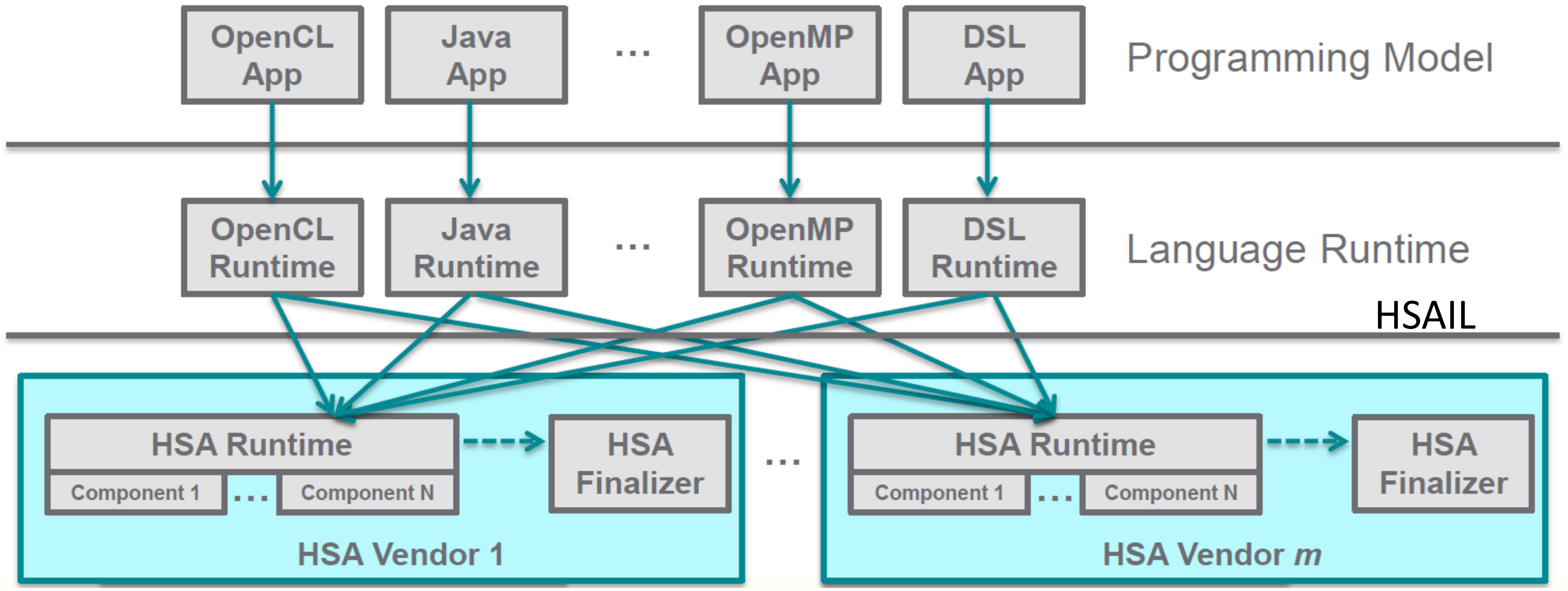
- HSA compilation stack
- HSA runtime stack



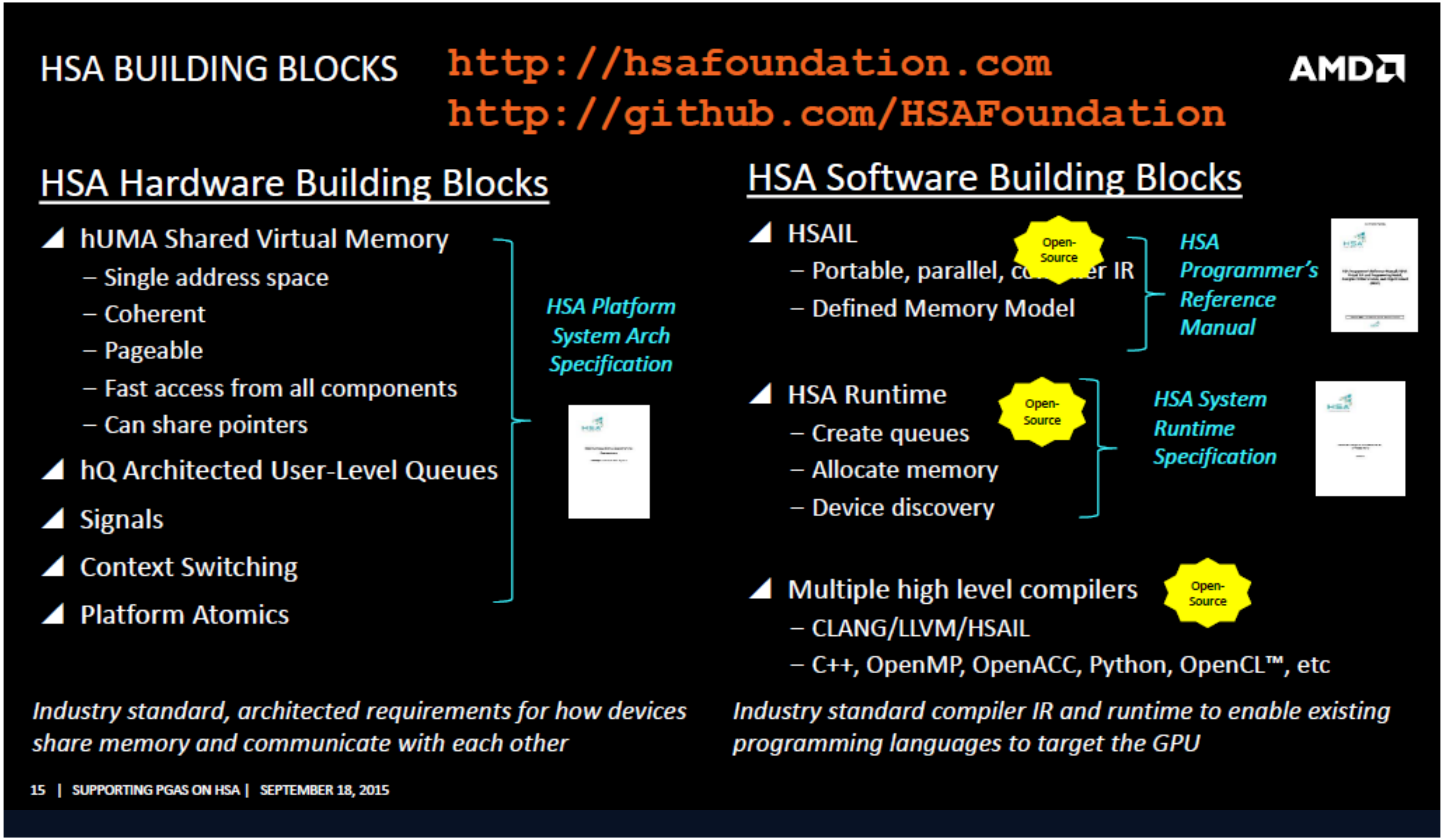


HSA software stack

- HSA supports many languages



Specifications and software



HSA BUILDING BLOCKS http://hsafoundation.com http://github.com/HSAFoundation

AMD

HSA Hardware Building Blocks

- hUMA Shared Virtual Memory
  - Single address space
  - Coherent
  - Pageable
  - Fast access from all components
  - Can share pointers
- hQ Architected User-Level Queues
- Signals
- Context Switching
- Platform Atomics

HSA Platform System Arch Specification

HSA Software Building Blocks

- HSAIL
  - Portable, parallel, c[illegible]er IR
  - Defined Memory Model
- HSA Runtime
  - Create queues
  - Allocate memory
  - Device discovery
- Multiple high level compilers
  - CLANG/LLVM/HSAIL
  - C++, OpenMP, OpenACC, Python, OpenCL™, etc

Open-Source

HSA Programmer's Reference Manual

Open-Source

HSA System Runtime Specification

Open-Source

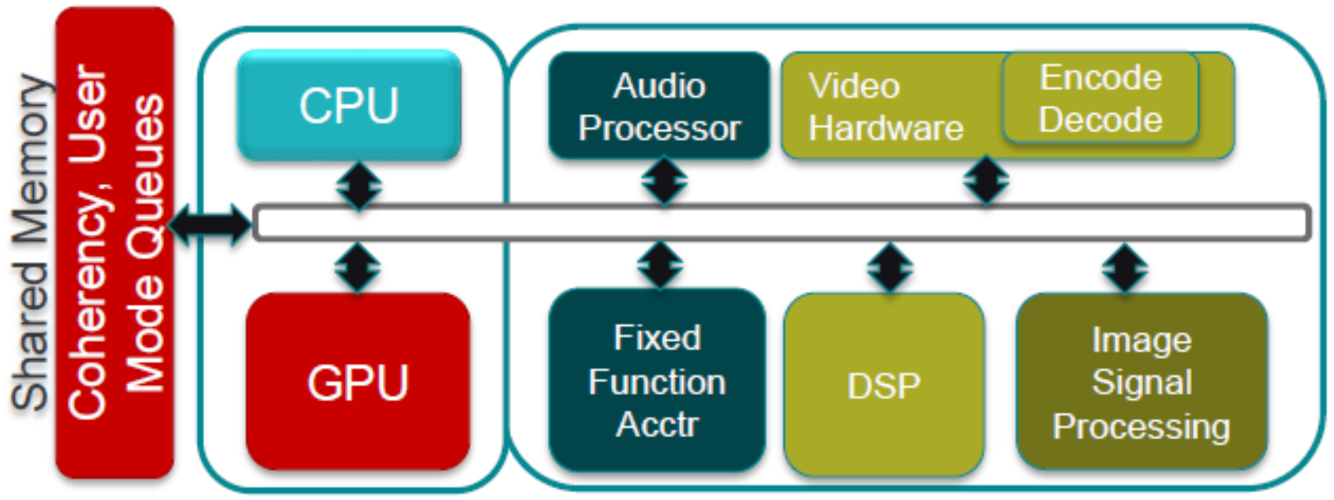
Industry standard, architected requirements for how devices share memory and communicate with each other

Industry standard compiler IR and runtime to enable existing programming languages to target the GPU

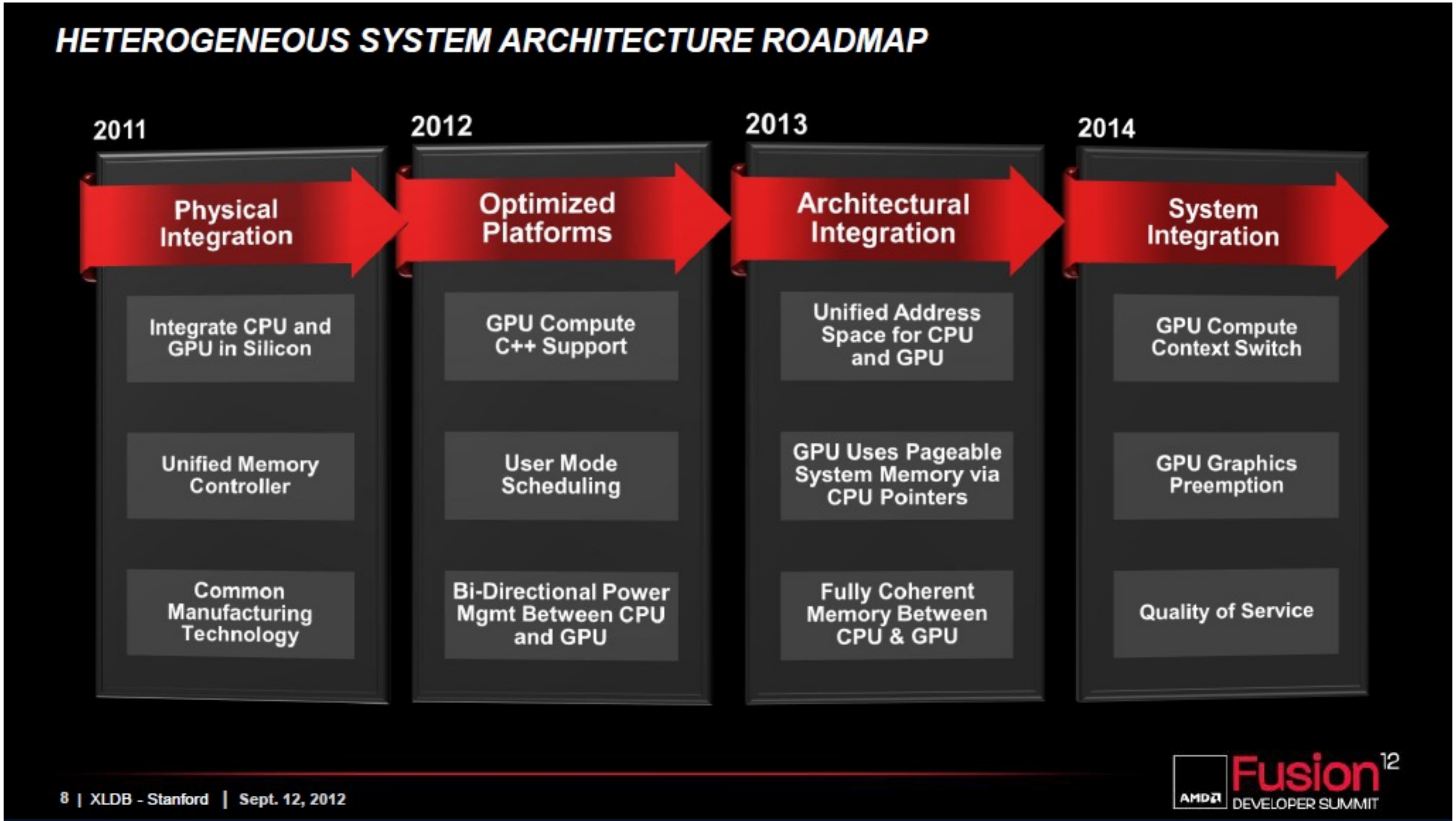
15 | SUPPORTING PGAS ON HSA | SEPTEMBER 18, 2015

HSA architecture V1

- GPU compute C++ support
- User Mode Scheduling
- Fully coherent memory between CPU & GPU
- GPU uses pageable system memory via CPU pointers
- GPU graphics pre-emption
- GPU compute context switch



AMD roadmaps



HETEROGENEOUS SYSTEM ARCHITECTURE ROADMAP

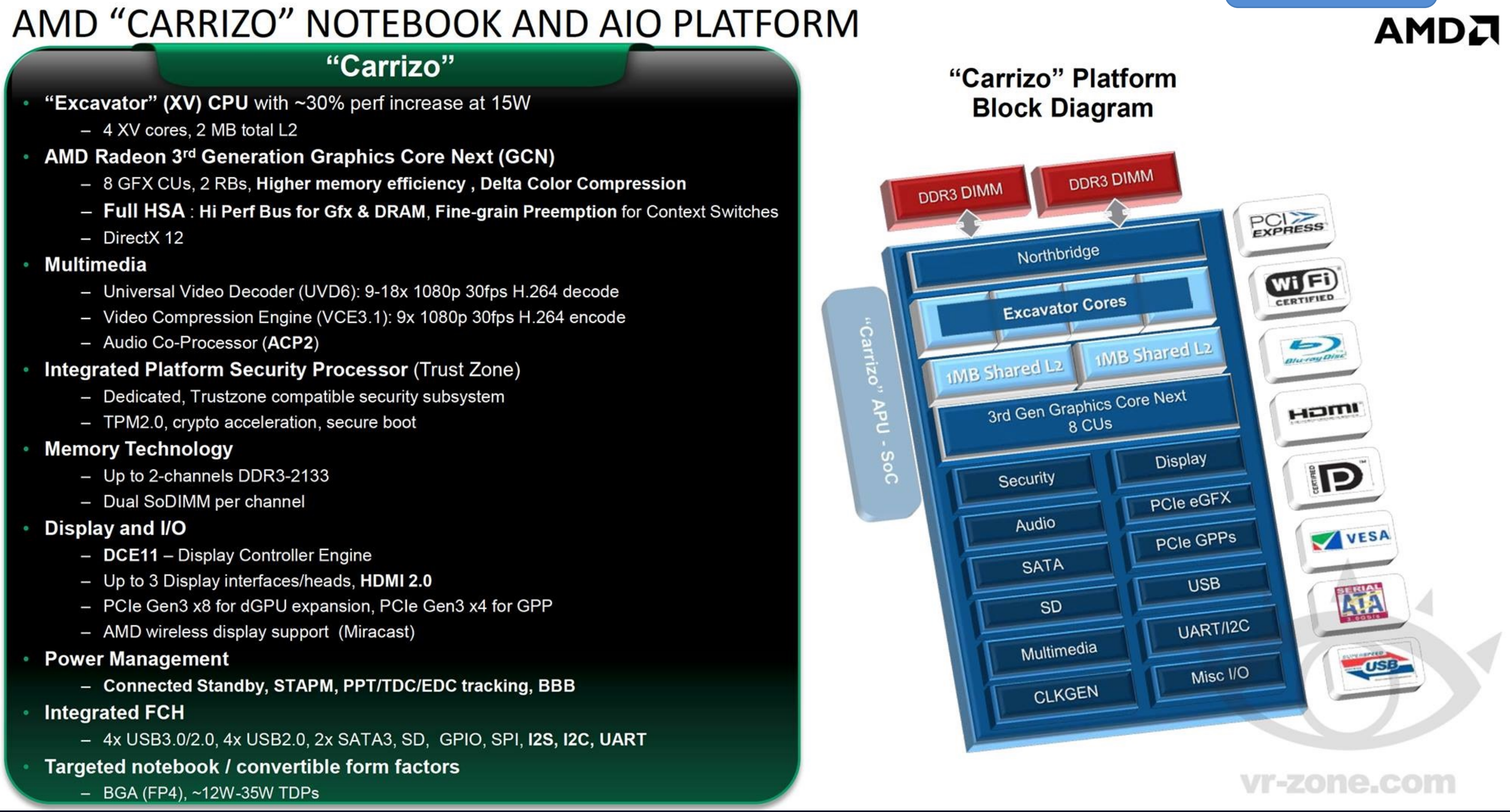
2011	2012	2013	2014
Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU and GPU in Silicon	GPU Compute C++ Support	Unified Address Space for CPU and GPU	GPU Compute Context Switch
Unified Memory Controller	User Mode Scheduling	GPU Uses Pageable System Memory via CPU Pointers	GPU Graphics Preemption
Common Manufacturing Technology	Bi-Directional Power Mgmt Between CPU and GPU	Fully Coherent Memory Between CPU & GPU	Quality of Service

8 | XDLB - Stanford | Sept. 12, 2012

AMD Fusion12 DEVELOPER SUMMIT

AMD roadmaps

2015



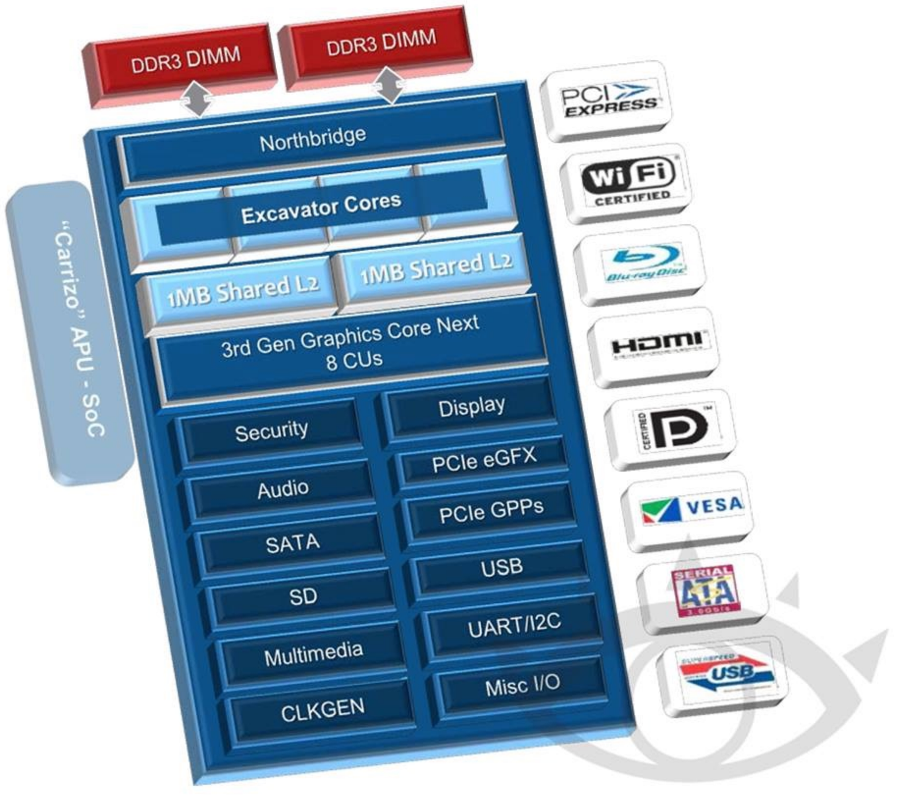
AMD "CARRIZO" NOTEBOOK AND AIO PLATFORM

AMD

"Carrizo"

- **"Excavator" (XV) CPU** with ~30% perf increase at 15W
  - 4 XV cores, 2 MB total L2
- **AMD Radeon 3rd Generation Graphics Core Next (GCN)**
  - 8 GFX CUs, 2 RBs, **Higher memory efficiency , Delta Color Compression**
  - **Full HSA** : **Hi Perf Bus for Gfx & DRAM**, **Fine-grain Preemption** for Context Switches
  - DirectX 12
- **Multimedia**
  - Universal Video Decoder (UVD6): 9-18x 1080p 30fps H.264 decode
  - Video Compression Engine (VCE3.1): 9x 1080p 30fps H.264 encode
  - Audio Co-Processor (**ACP2**)
- **Integrated Platform Security Processor** (Trust Zone)
  - Dedicated, Trustzone compatible security subsystem
  - TPM2.0, crypto acceleration, secure boot
- **Memory Technology**
  - Up to 2-channels DDR3-2133
  - Dual SoDIMM per channel
- **Display and I/O**
  - **DCE11** – Display Controller Engine
  - Up to 3 Display interfaces/heads, **HDMI 2.0**
  - PCIe Gen3 x8 for dGPU expansion, PCIe Gen3 x4 for GPP
  - AMD wireless display support (Miracast)
- **Power Management**
  - **Connected Standby, STAPM, PPT/TDC/EDC tracking, BBB**
- **Integrated FCH**
  - 4x USB3.0/2.0, 4x USB2.0, 2x SATA3, SD, GPIO, SPI, **I2S, I2C, UART**
- **Targeted notebook / convertible form factors**
  - BGA (FP4), ~12W-35W TDPs

"Carrizo" Platform Block Diagram


DDR3 DIMM
DDR3 DIMM
"Carrizo" APU - SoC
Northbridge
Excavator Cores
1MB Shared L2
1MB Shared L2
3rd Gen Graphics Core Next
8 CUs
Security
Display
Audio
PCIe eGFX
SATA
PCIe GPPs
SD
USB
Multimedia
UART/I2C
CLKGEN
Misc I/O
PCI EXPRESS
Wi Fi CERTIFIED
Blu-ray Disc
HDMI
DP
VESA
SERIAL ATA
USB