Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

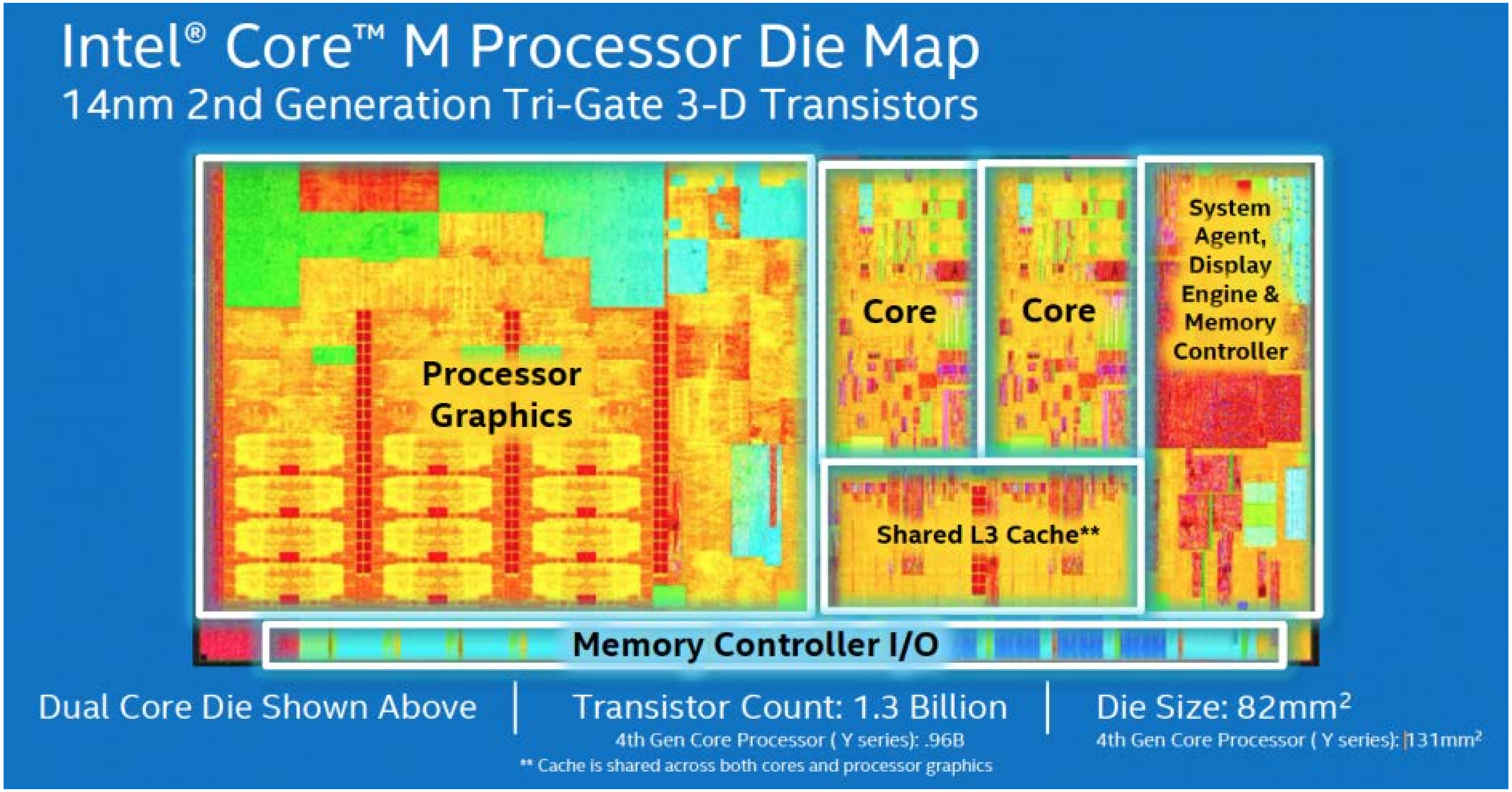
Department of Information Technology
and Electrical Engineering

Heterogeneous Architecture

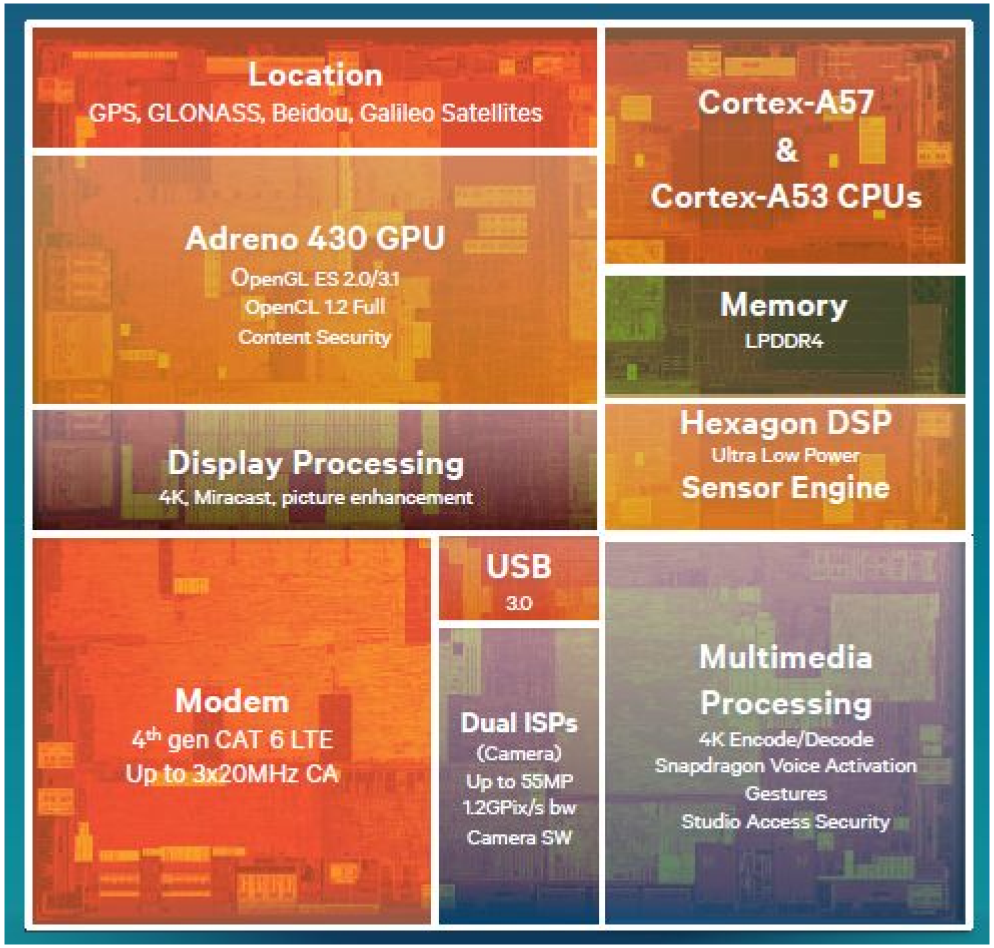
Luca Benini

lbenini@iis.ee.ethz.ch

Intel’s Broadwell


Intel® Core™ M Processor Die Map
14nm 2nd Generation Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
System Agent, Display Engine & Memory Controller
Shared L3 Cache**
Memory Controller I/O
Dual Core Die Shown Above
Transistor Count: 1.3 Billion
4th Gen Core Processor (Y series): .96B
Die Size: 82mm²
4th Gen Core Processor (Y series): 131mm²
** Cache is shared across both cores and processor graphics

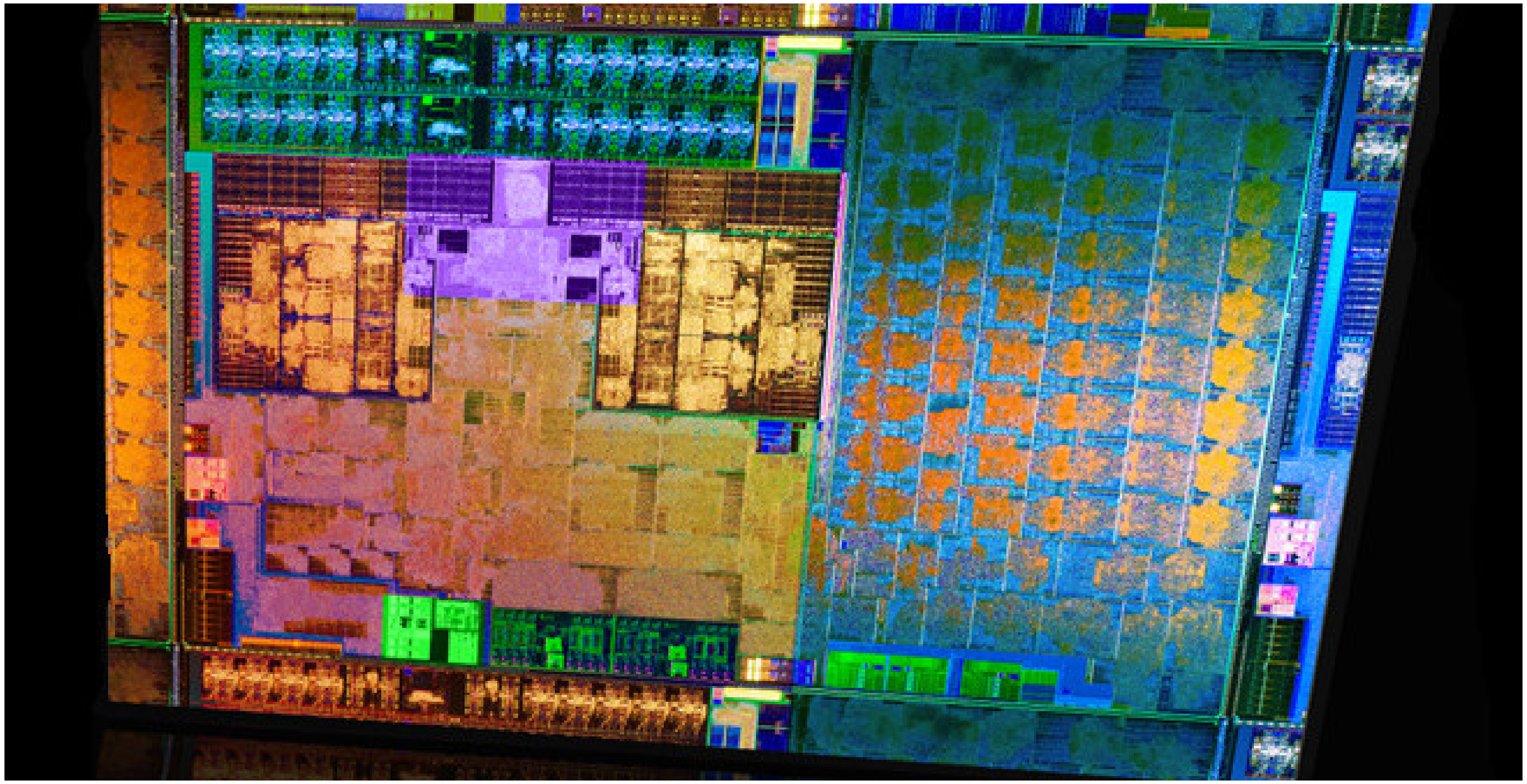
Qualcomm’s Snapdragon 810



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Information Technology
and Electrical Engineering

AMD Bristol Ridge

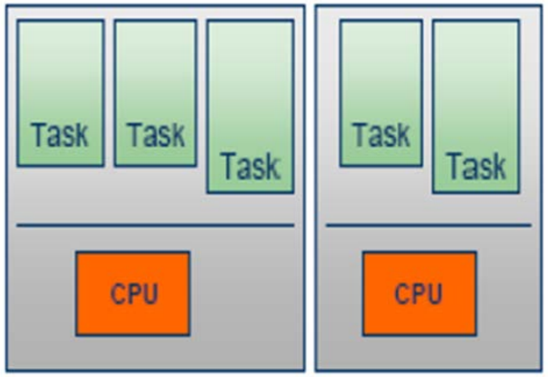


Heterogeneous Multicores

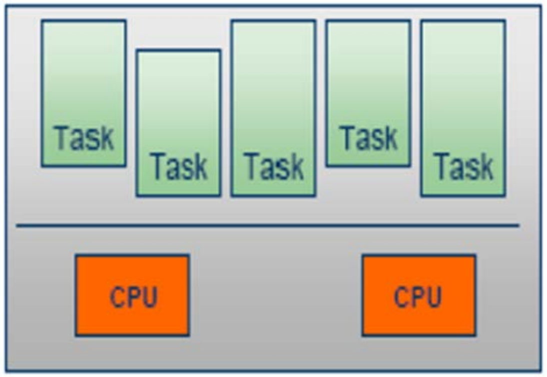
Deszo Sima (Univ. Budapest)

Vocabulary in the multi-era

- AMP (Asymmetric MP)
  - Each processor has local memory
  - Tasks statically allocated to one processor

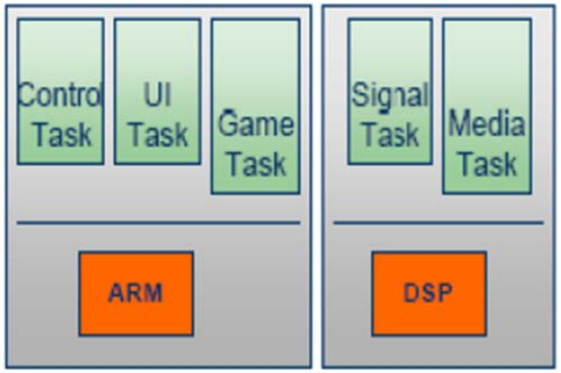


- SMP (Symmetric MP)
  - Processors share memory
  - Tasks dynamically scheduled to any processor

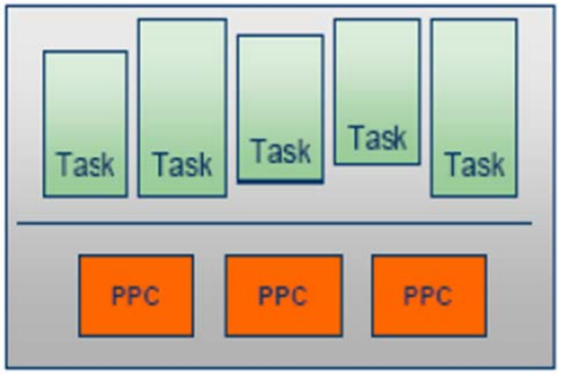


Vocabulary in the multi-era

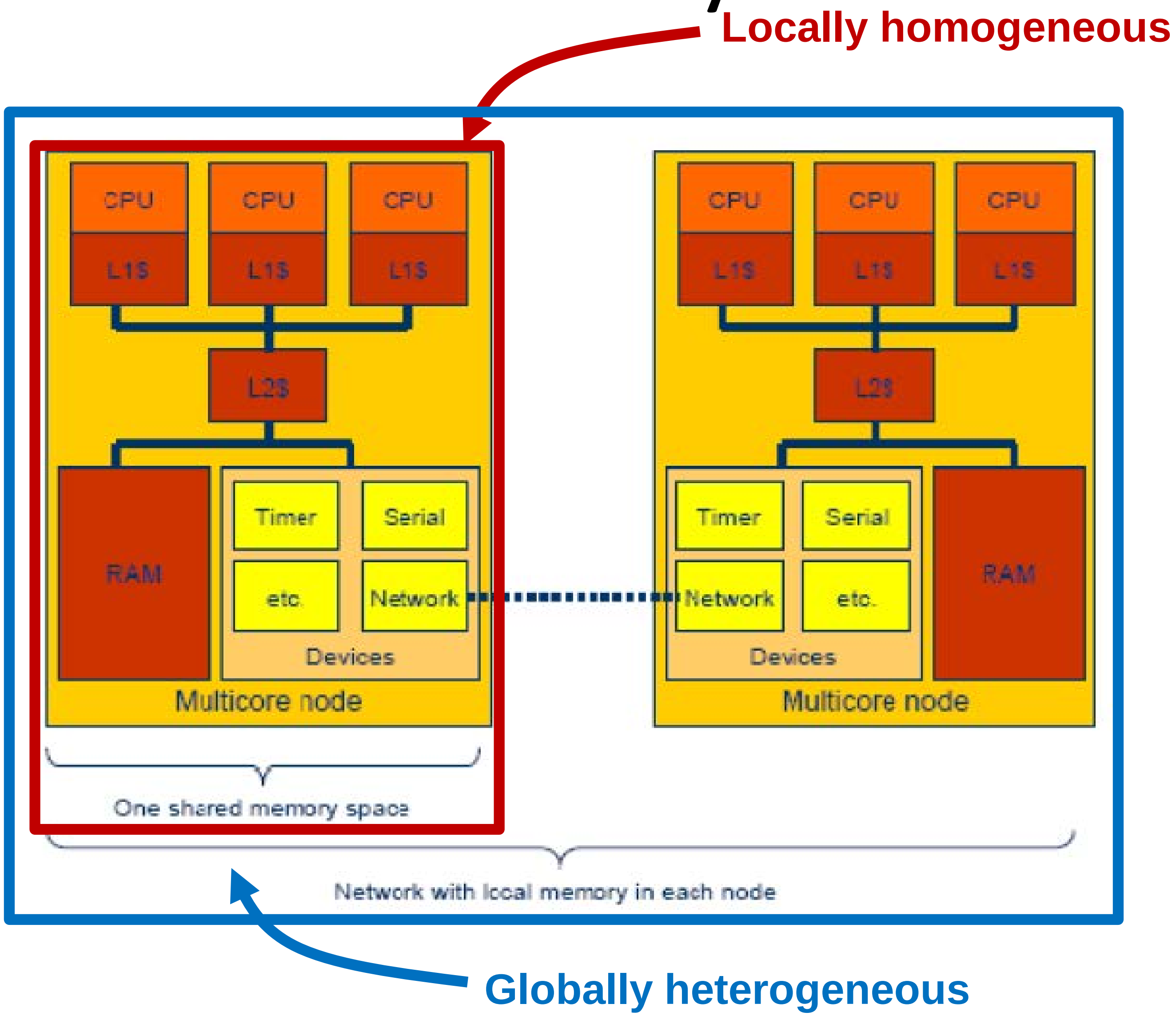
- Heterogeneous:
  - Specialization among processors
  - Often different instruction sets
  - Usually AMP design



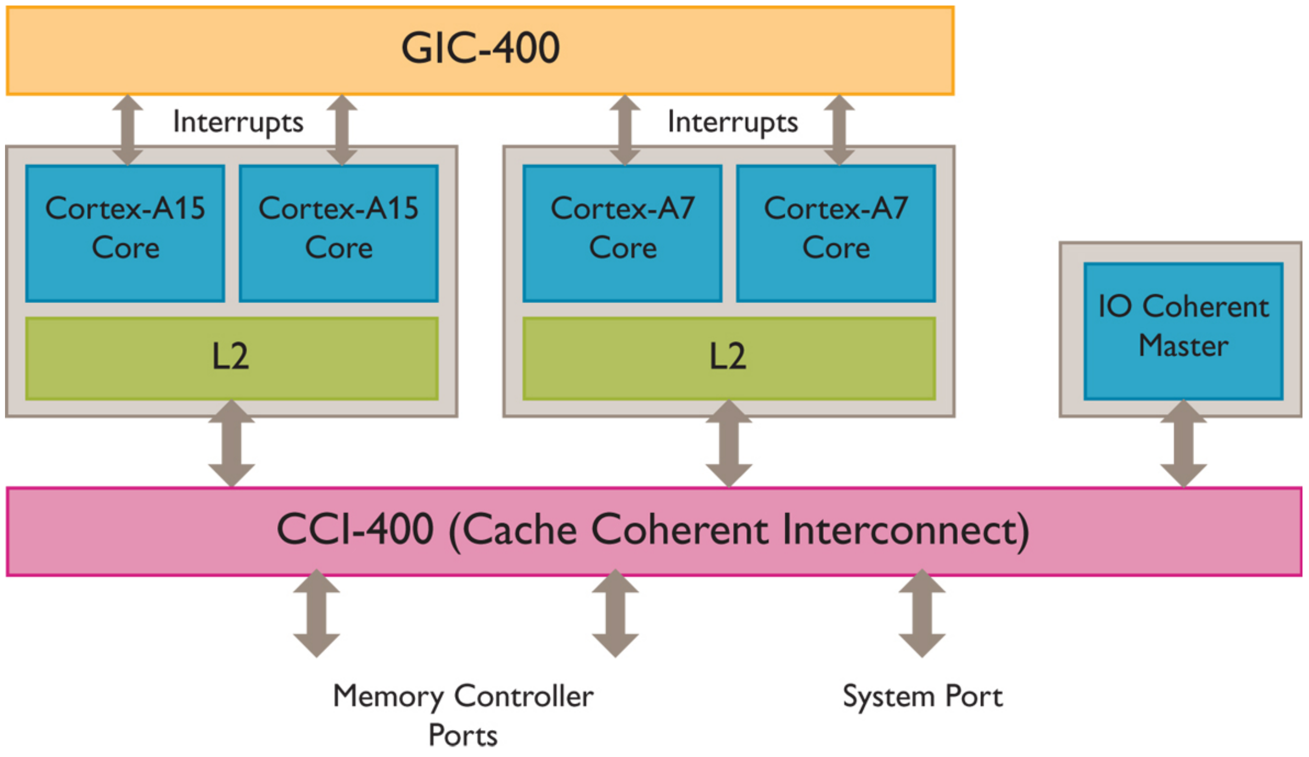
- Homogeneous:
  - All processors have the same instruction set
  - Processors can run any task
  - Usually SMP design



Future many-cores

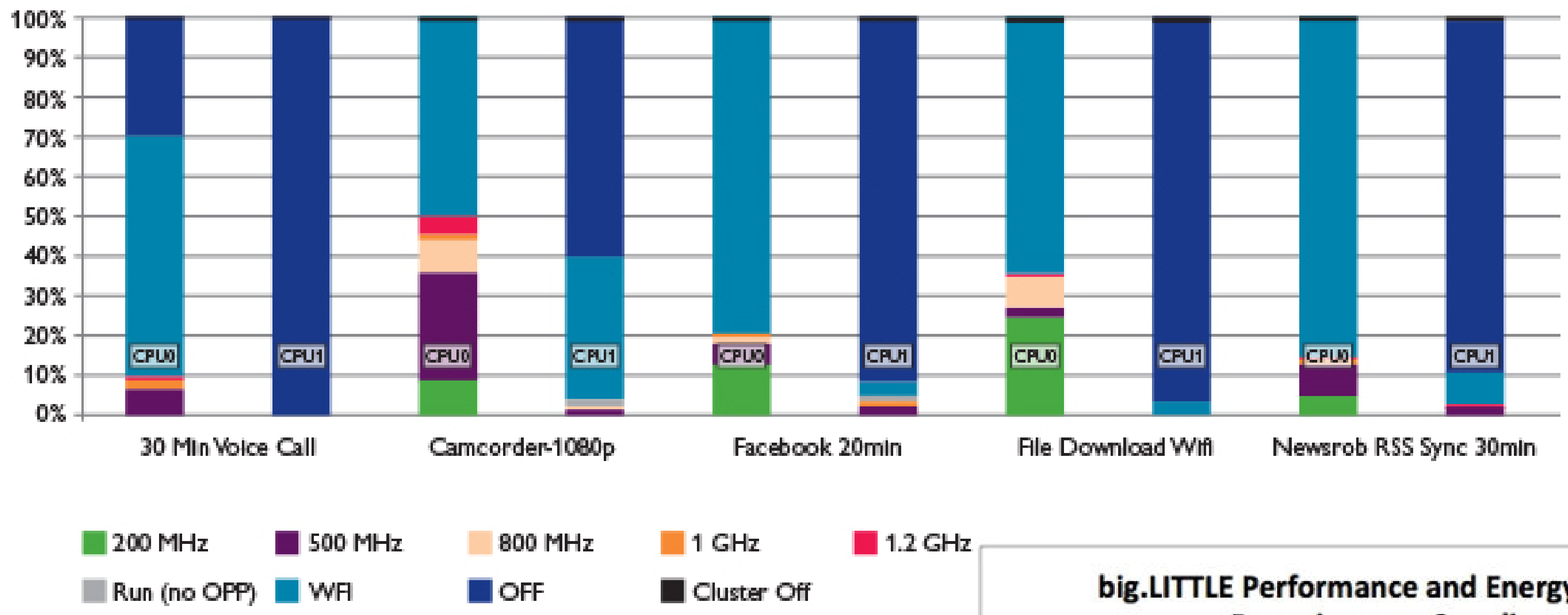


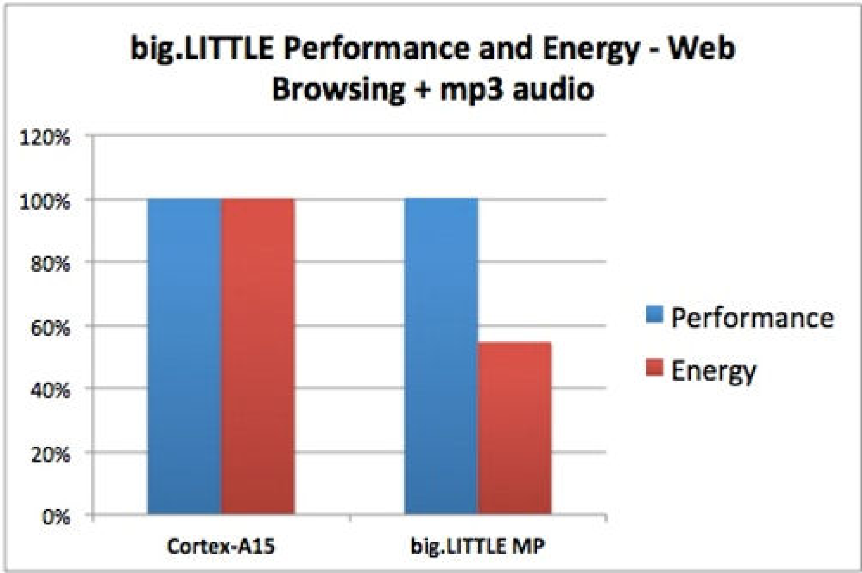
ARM big.LITTLE



- DVFS
- Task Migration
- SW Overhead
- Memory coherency

ARM big.LITTLE Energy Saving





4.3 Principle of ARM's big.LITTLE technology (3)

Exclusive/inclusive use of the clusters

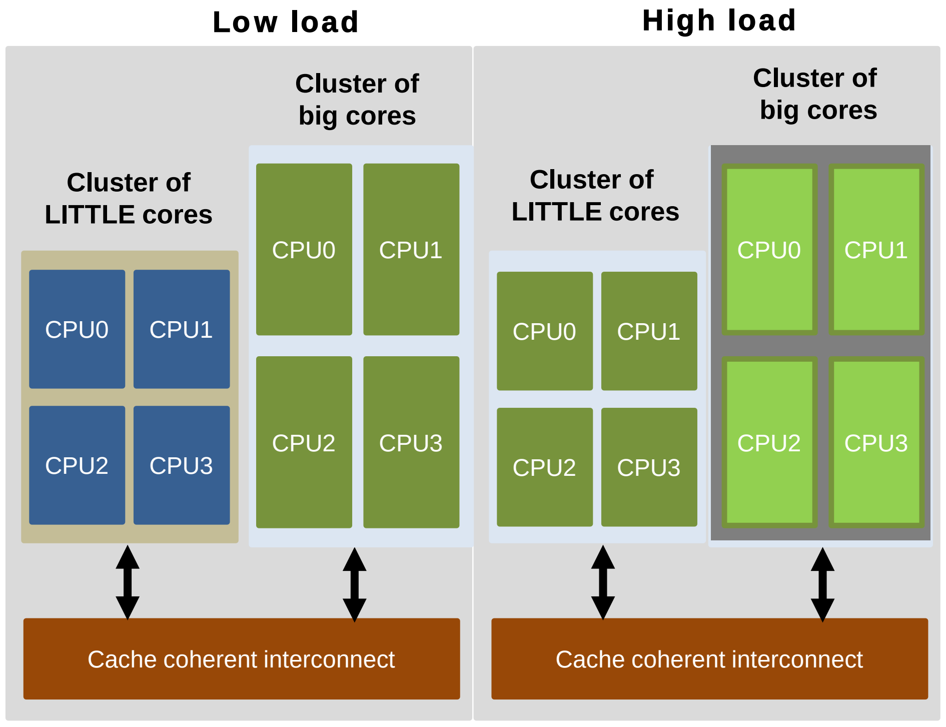
Exclusive / inclusive use of the clusters

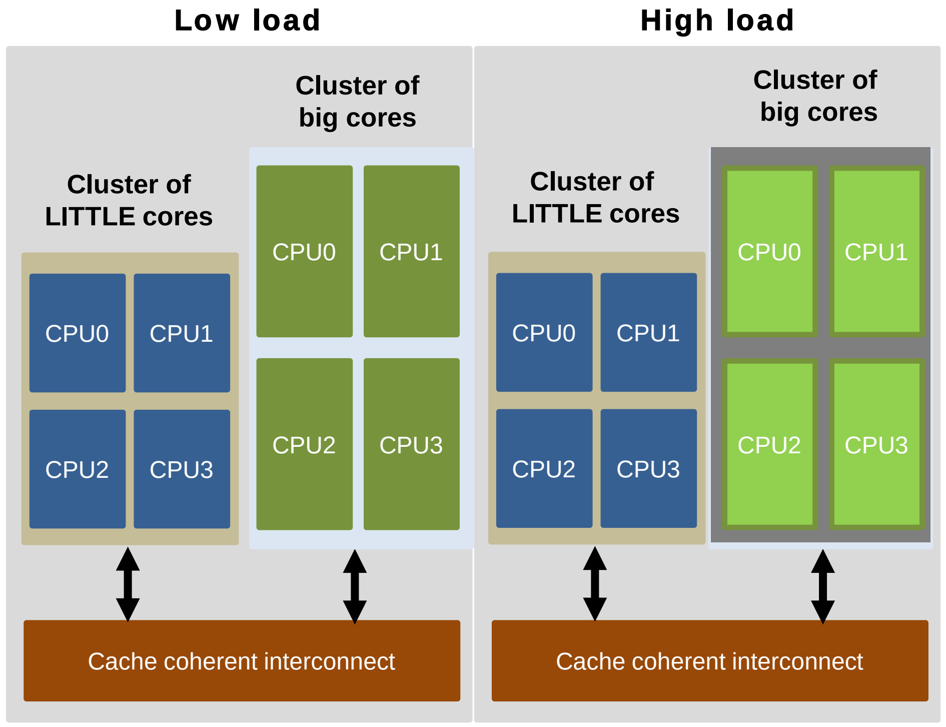
Exclusive use of the clusters

Clusters are used exclusively, i.e. at a time one of the clusters is in use as shown below for the cluster migration model (to be discussed later)

Inclusive use of the clusters

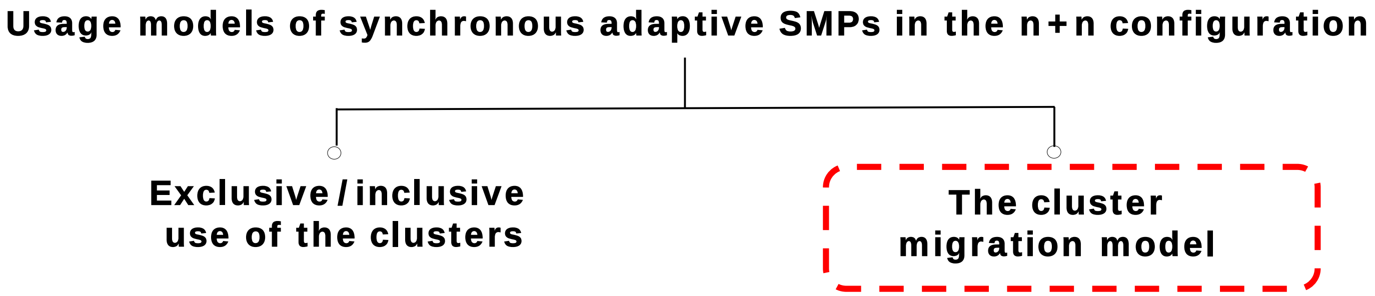
Clusters are used inclusively, i.e. at a time both clusters can be used partly or entirely





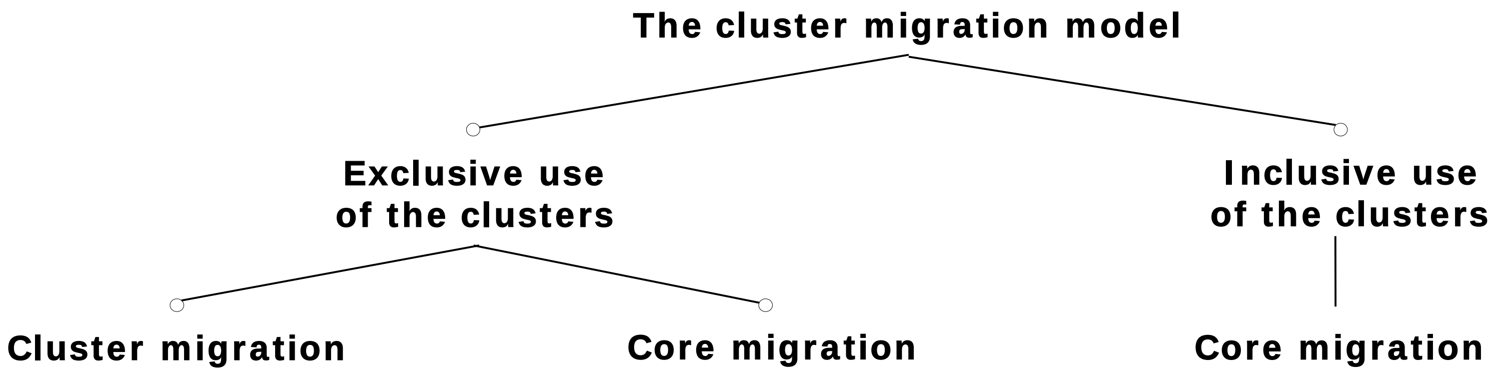
4.3 Principle of ARM's big.LITTLE technology (2)

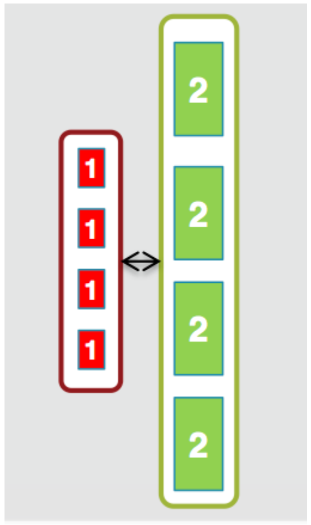
Usage models of synchronous adaptive SMPs in the n + n configuration


Usage models of synchronous adaptive SMPs in the n+n configuration
Exclusive / inclusive use of the clusters
The cluster migration model

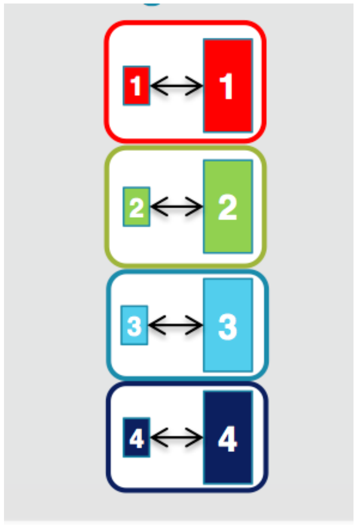
4.3 Principle of ARM’s big.LITTLE technology (4)

The cluster migration model [5]

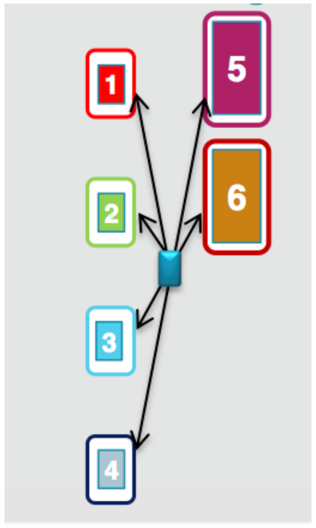

The cluster migration model
Exclusive use of the clusters
Inclusive use of the clusters
Cluster migration
Core migration
Core migration



big.LITTLE processing with cluster migration



big.LITTLE processing with core migration

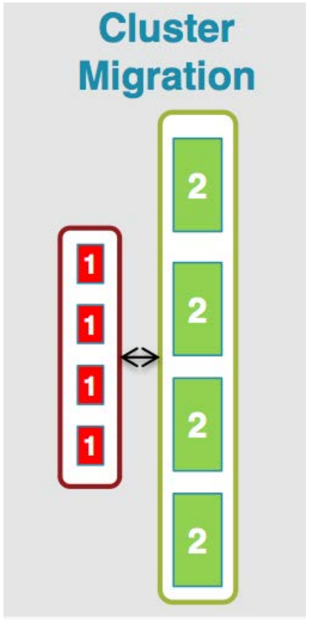


big.LITTLE MP

4.3 Principle of ARM's big.LITTLE technology (5)

Big.LITTLE processing with cluster migration [5]

- There are two core clusters, the LITTLE core cluster and the big core cluster.
- Tasks run on either the LITTLE or the big core cluster, so only one core cluster is active at any time (except a short interval during a cluster switch).
- Low workloads, such as background synch tasks, audio or video playback run typically on the LITTLE core cluster.
- If the workload becomes higher than the max performance of the LITTLE core cluster the workload will be migrated to the big core cluster and vice versa.



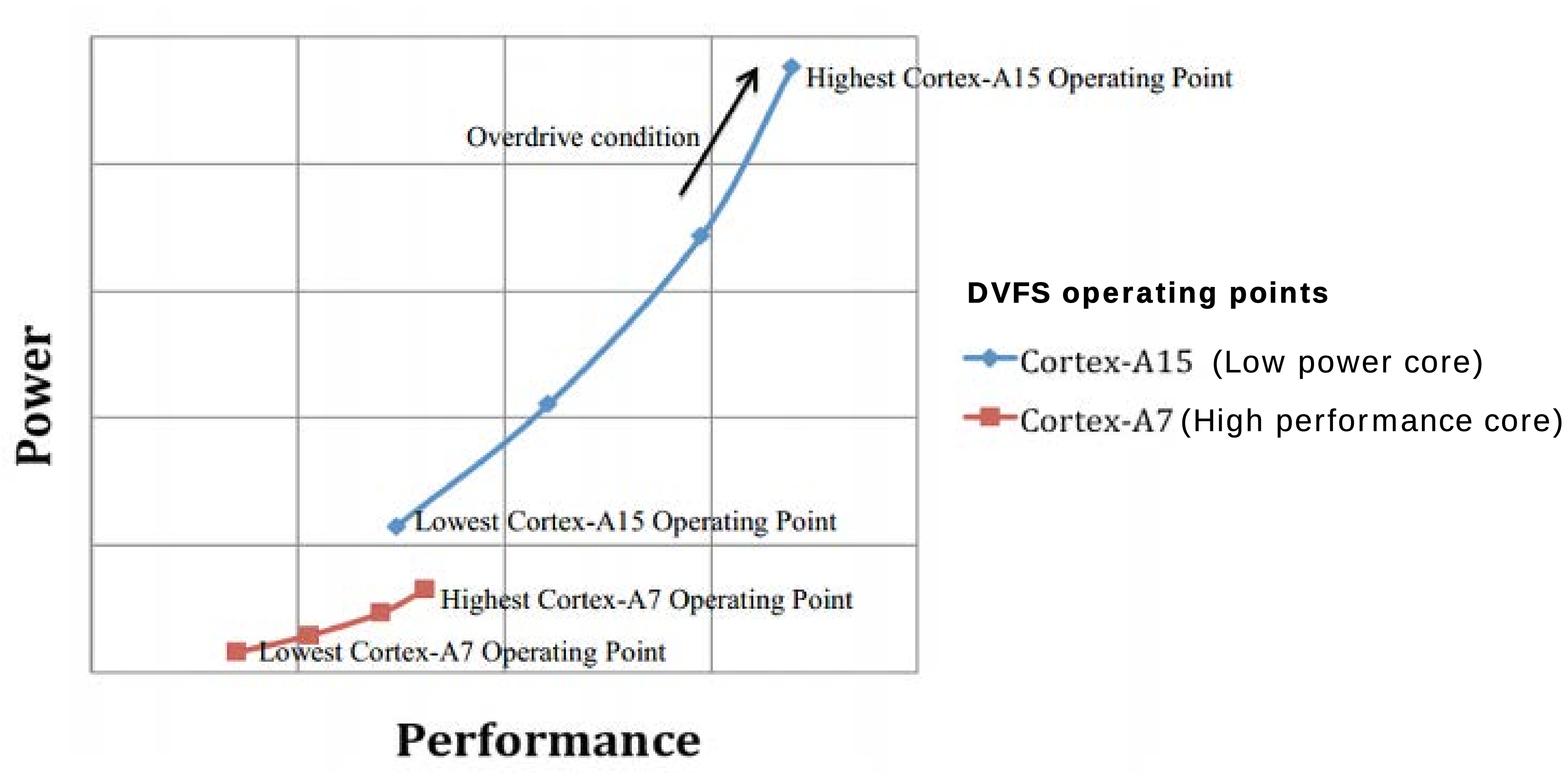
4.3 Principle of ARM's big.LITTLE technology (6)

Cluster switches [6]

- Cluster selection is driven by OS power management.
- OS (e.g. the Linux cpufreq routine) samples the load for all cores in the cluster and selects an operating point for the cluster.
- It switches clusters at terminal points of the current clusters DVFS curve, as illustrated in the next Figure.

4.3 Principle of ARM's big.LITTLE technology (7)

Power/performance curve during cluster switching [7]



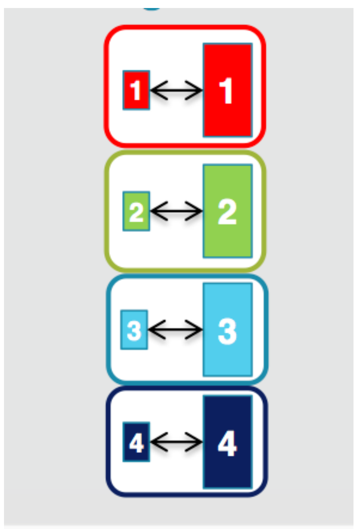
- A switch from the low power cluster to the high performance cluster is an extension of the DVFS strategy.
- A cluster switch lasts about 30 kcycles.

4.3 Principle of ARM’s big.LITTLE technology (8)

Big.LITTLE processing with core migration [5], [8]

- There are two core clusters, the LITTLE core cluster and the big core cluster.
- Cores are grouped into pairs of one big core and one LITTLE core.

  The LITTLE and the big core of a group are used exclusively.
- Each LITTLE core can switch to its big counterpart if it meets a higher load than its max. performance and vice versa.
- Each core switch is independent from the others.



4.3 Principle of ARM's big.LITTLE technology (9)

Core switches [6]

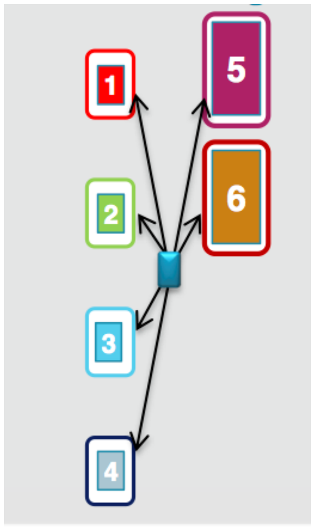
- Core selection in any core pair is performed by OS power management.
- The DVFS algorithm monitors the core load.

  When a LITTLE core cannot service the actual load, a switch to its big counterpart is initiated and the LITTLE core is turned off and vice versa.

4.3 Principle of ARM's big.LITTLE technology (10)

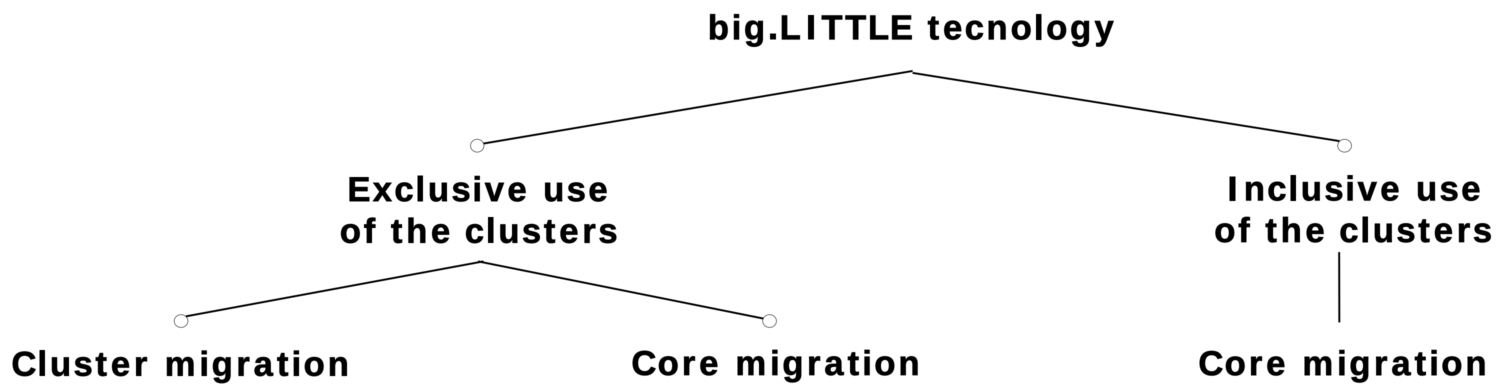
big.LITTLE MP processing with core migration [8],[5]

- The OS scheduler has all cores of both clusters at its disposal and can activate all cores at any time.
- Tasks can run or be moved between the LITTLE cores and the big cores as decided by the scheduler.
- big.LITTLE MP termed also as Heterogeneous Multiprocessing (HMP).



4.3 Principle of ARM's big.LITTLE technology (11)

Use of the big.LITTLE technology in recent mobile processors



big.LITTLE processing with cluster migration

Described first in ARM's White Paper (2011) [3]

Used in

Samsung Exynos 5 Octa 5410 (2013) (4 + 4 cores)

big.LITTLE processing with core migration

Described first in ARM's White Paper (2012) [9]

big.LITTLE MP (Heterogeneous Multiprocessing)

Described first in ARM's White Paper (2011) [3]

Samsung HMP on Exynos 5 Octa 5420 (2013) (4 + 4 cores)

Mediatek MT 8135 (2013) (2 + 2 cores)

Renesas MP 6530 (2013) (2 + 2 cores)

CPU+GPU integration

Deszo Sima (Univ. Budapest)

1. Introduction to architectural integration of the CPU and GPU (1)

1. Introduction to architectural integration of the CPU and GPU

Aim of architectural integration of the CPU and GPU

Accelerated graphics processing

- Accelerating graphics processing
  by the higher bandwidth
  of connecting the CPU and the GPU.
- Cost reduction

Heterogeneous processing

- Accelerating HPC by the GPU,
  i.e. by the large number of FP units
  available in a GPU.

Needs HLL support (CUDA, OPenCL)

1. Introduction to architectural integration of the CPU and GPU (2)

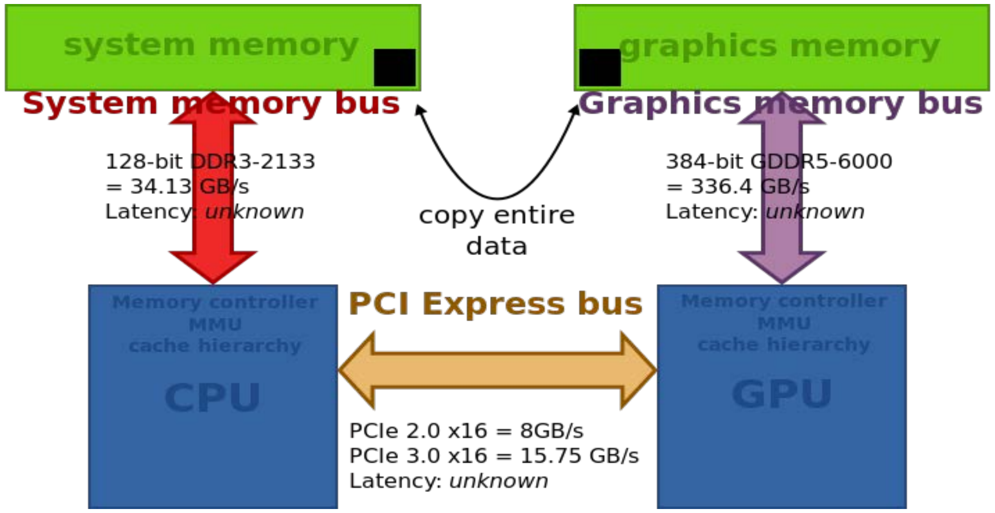
Implementation alternatives of graphics memory [1]

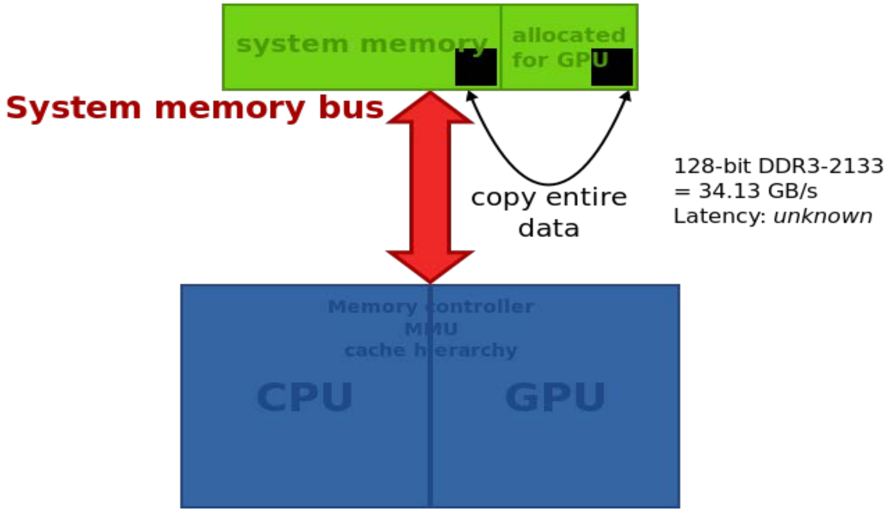
Implementation of the graphics memory

Discrete graphics memory

Unified system memory
(Shared memory)
(Unified Virtual Memory)
(Unified Memory Architecture)

Implementation example





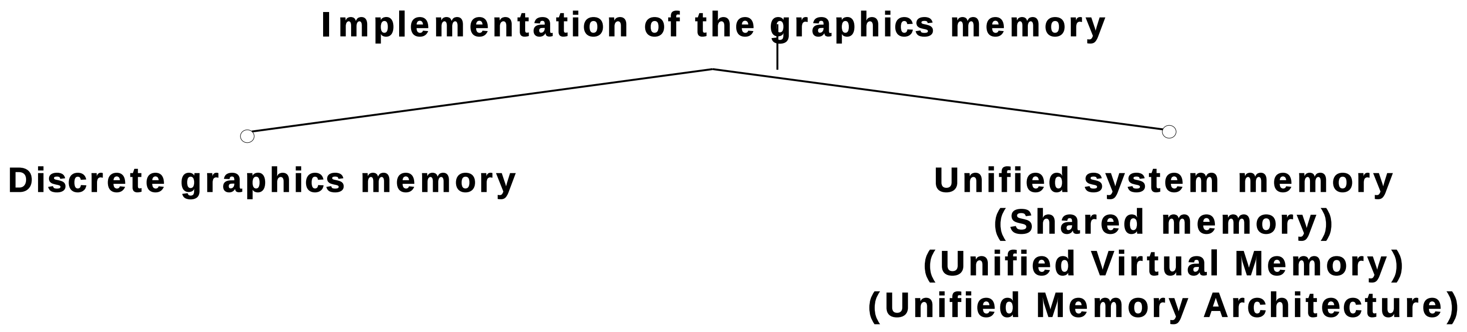
Typ. use

Graphics cards

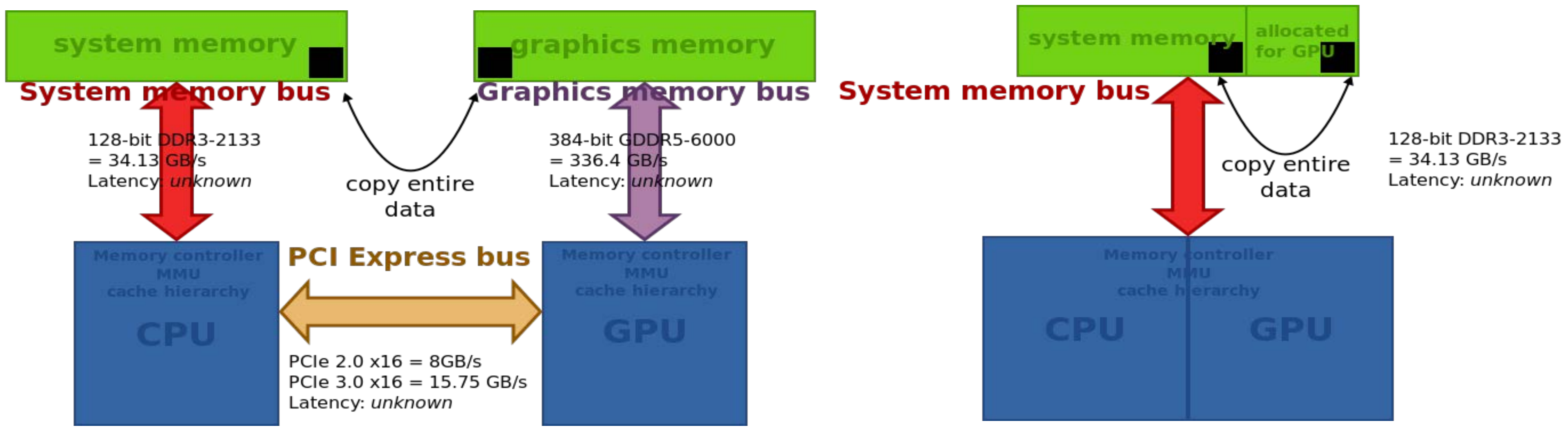
On-die integrated graphics

1. Introduction to architectural integration of the CPU and GPU (3)

Key benefit / drawback of the Unified system memory [1]



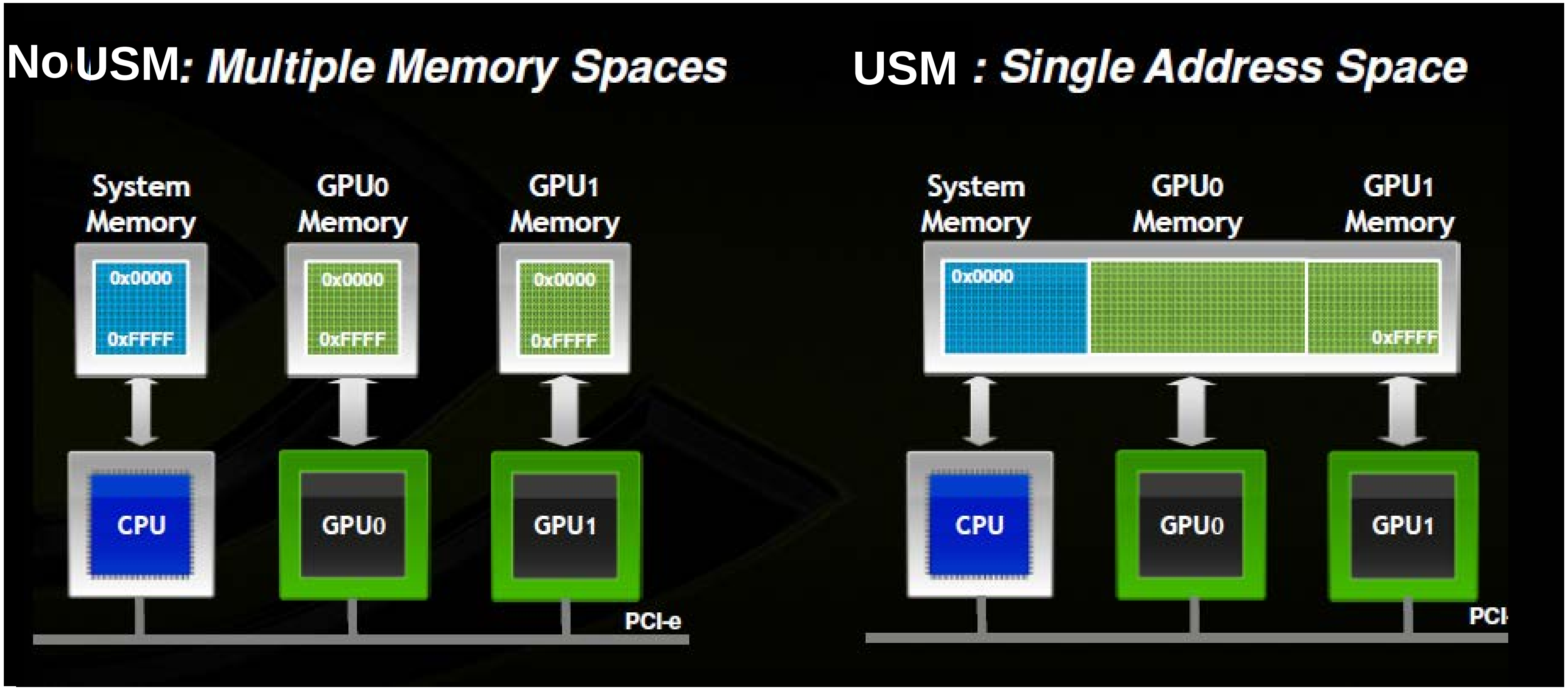
Implementation example



It eliminates the need for an extra graphics memory controller and bus but has the constraints of a reduced memory bandwidth.

1. Introduction to architectural integration of the CPU and GPU (4)

Address spaces for Discrete and Unified System Memory (USM) [2]



1. Introduction to architectural integration of the CPU and GPU (5)

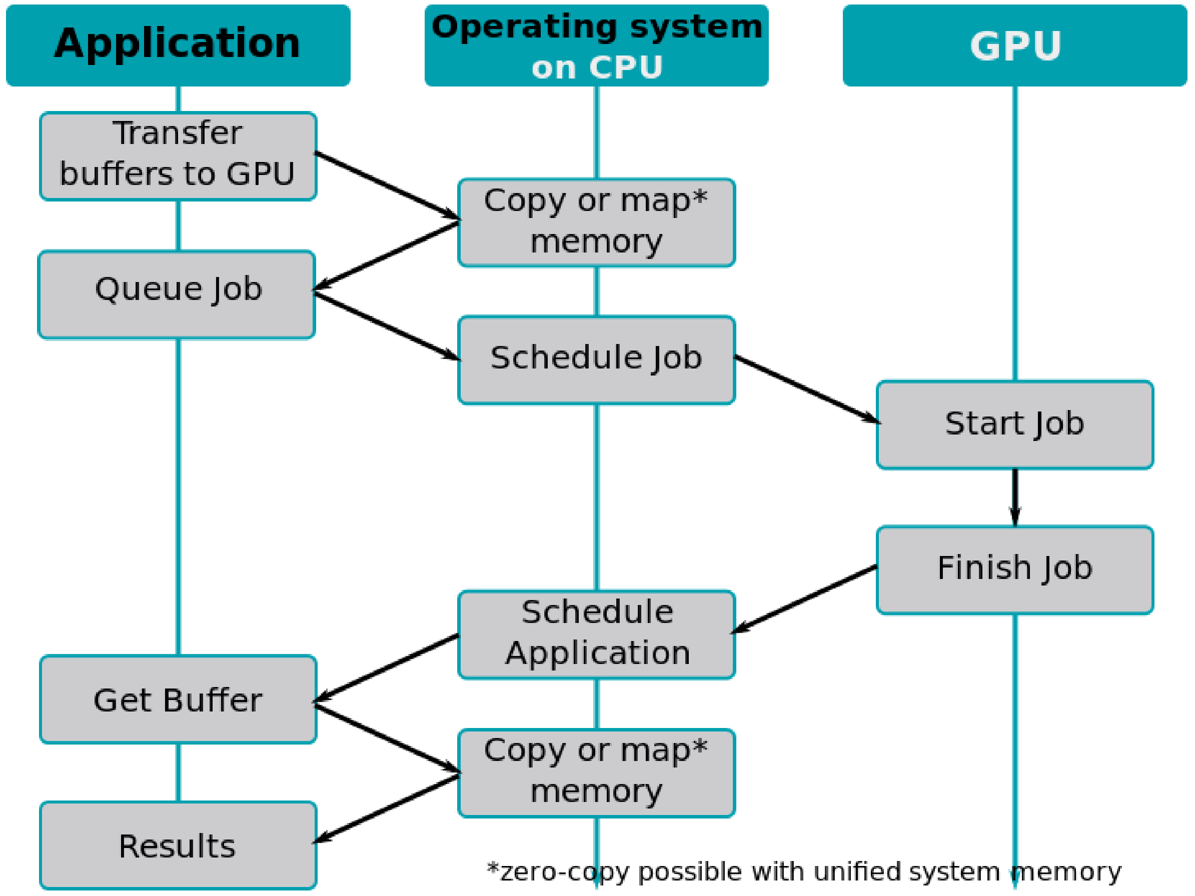
Main difficulty of heterogeneous computing

- GPUs operate from their own address spaces, thus without appropriate hardware and software support data to be processed by GPUs need to be loaded into their address space and the results need to be sent back to the host.
- Transferring data may be avoided with unified system memory and suitable software support (CUDA 4.0 or OpenCL 1.2 or higher) assuming appropriate hardware.

  Then data transfer will be substituted by address mapping (called zero copy).

1. Introduction to architectural integration of the CPU and GPU (6)

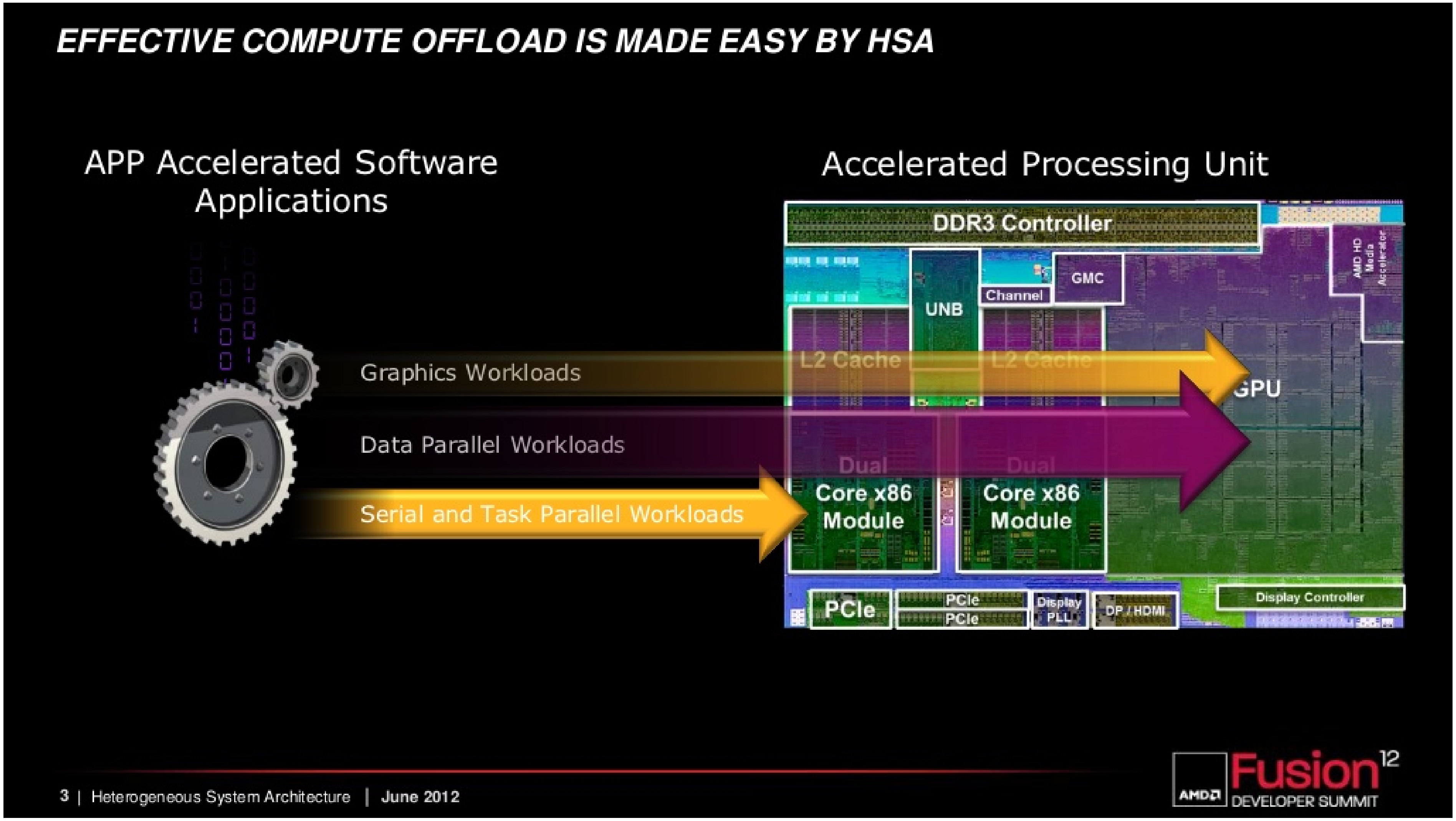
Tasks to be performed for copying data / mapping address spaces for GPUs [1]



2. AMD’s approach to support heterogeneous computing

2.1 Introduction to HSA (2)

Aim of HSA-1 [4]



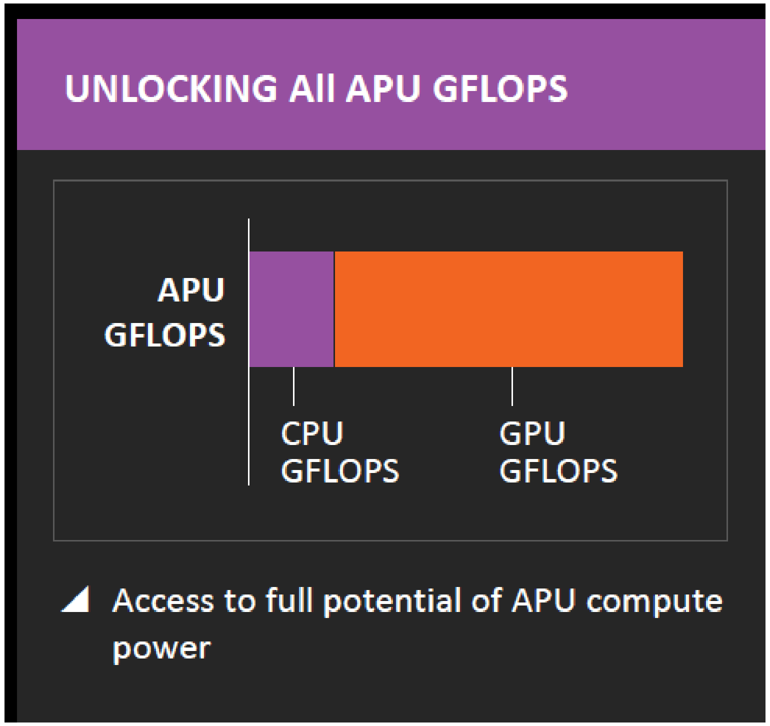
EFFECTIVE COMPUTE OFFLOAD IS MADE EASY BY HSA

3 | Heterogeneous System Architecture | June 2012

AMD Fusion 12 DEVELOPER SUMMIT

2.1 Introduction to HSA (3)

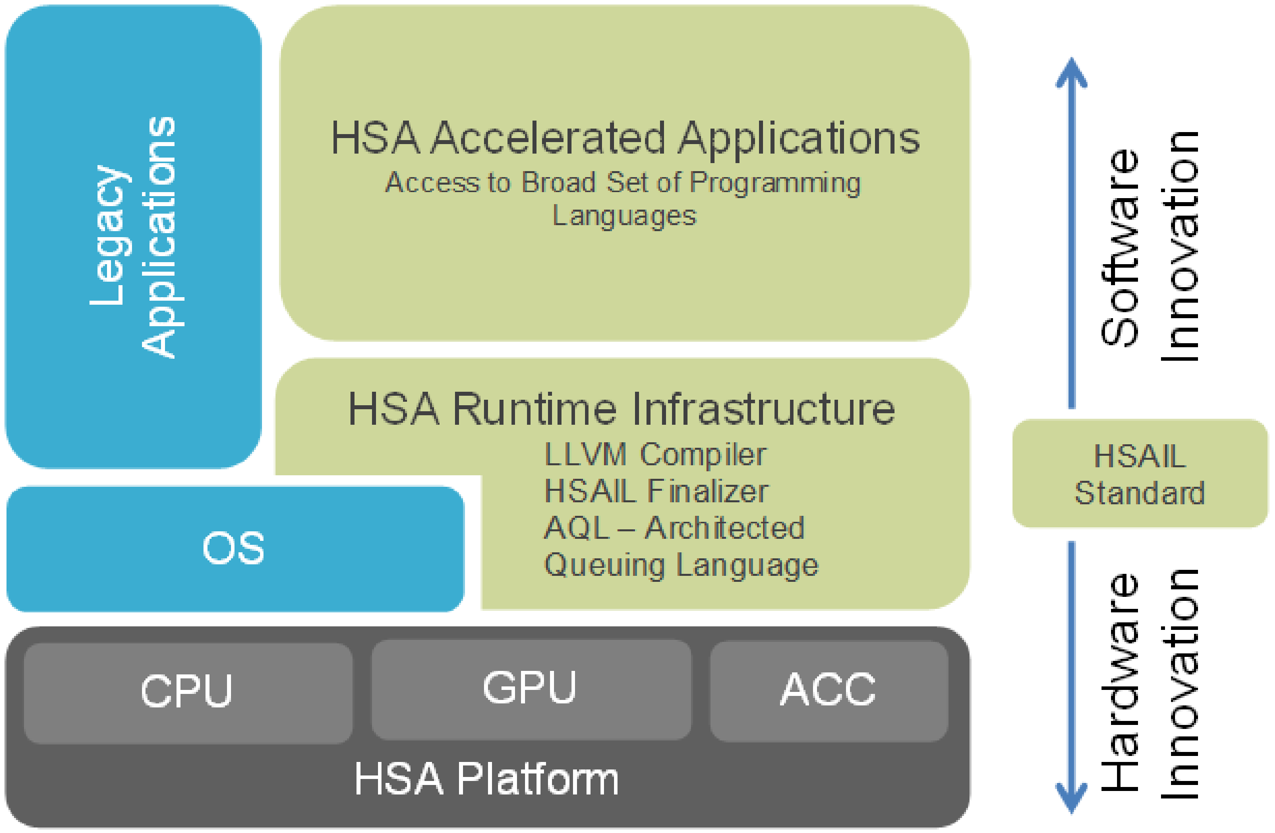
Aim of HSA-2 [5]



◢ Access to full potential of APU compute power

2.1 Introduction to HSA (4)

The entire HSA solution stack [6]


Legacy Applications
HSA Accelerated Applications
Access to Broad Set of Programming Languages
HSA Runtime Infrastructure
LLVM Compiler
HSAIL Finalizer
AQL – Architected Queuing Language
OS
CPU
GPU
ACC
HSA Platform
Software Innovation
HSAIL Standard
Hardware Innovation

HSAIL: HSA Intermediate Language

2.1 Introduction to HSA (5)

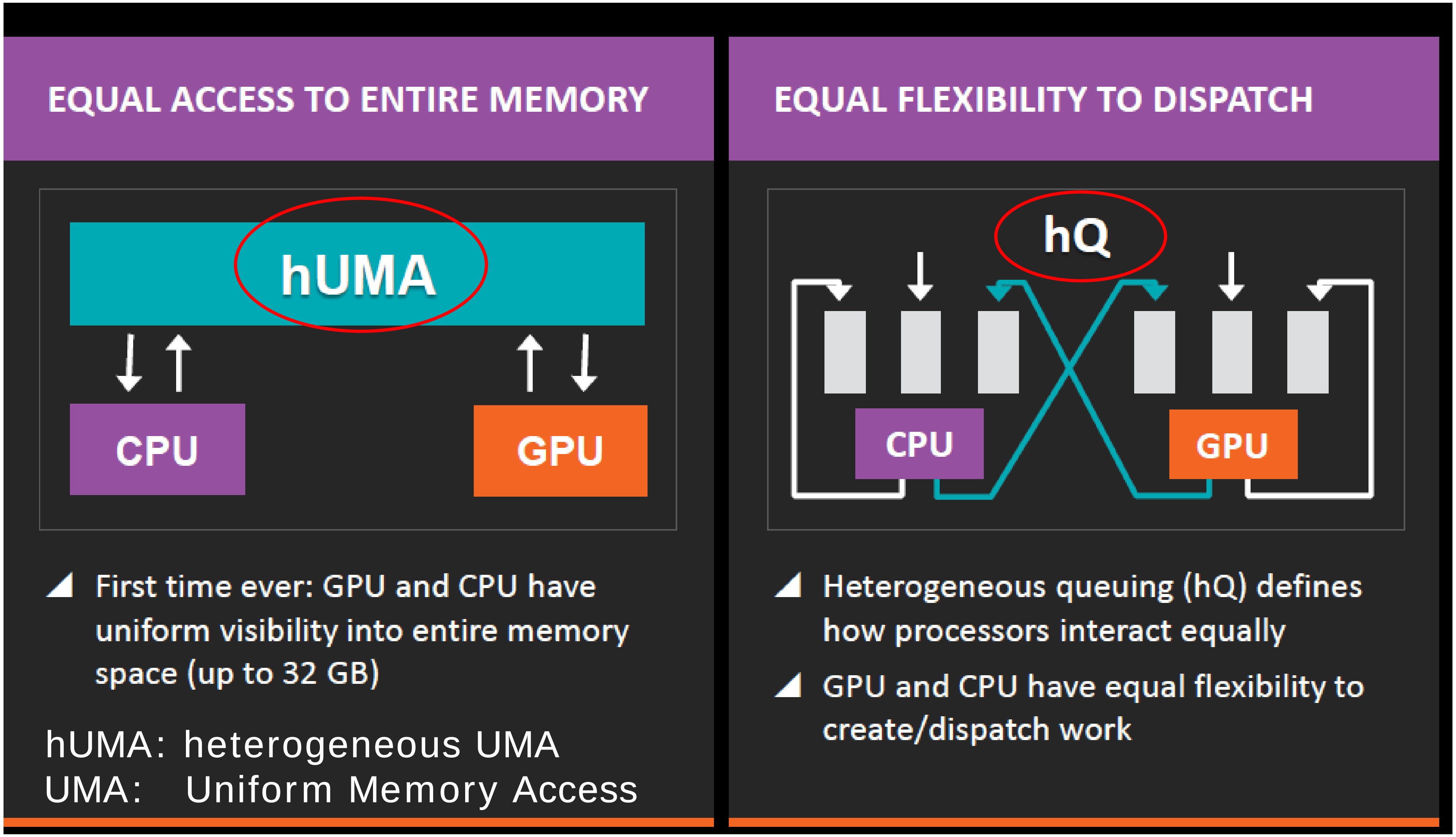
Establishment of the HAS Foundation

- In 6/2012 AMD, ARM, TI Qualcomm, Samsung and several other leading semiconductor firms established the Heterogeneous System Architecture (HSA) Foundation.
- It is a non-profit consortium that aims at defining and promoting open standards for heterogeneous computing.

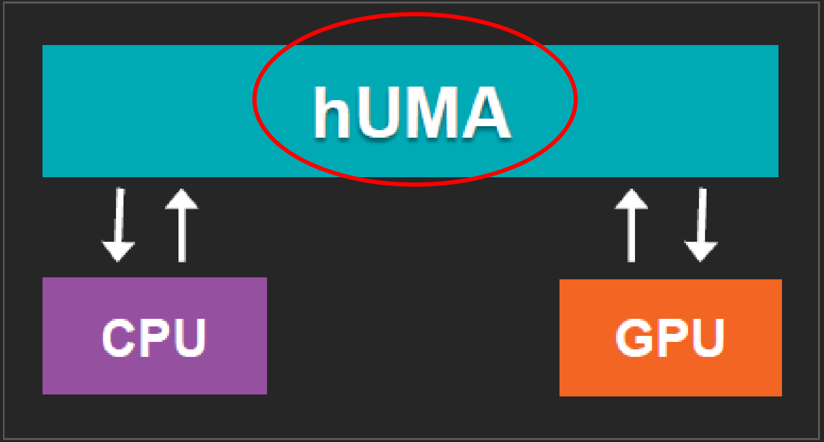
2.2 Key hardware enhancements needed to implement HSA

2.2 Key hardware enhancements needed to implement HSA (1)

2.2 Key hardware enhancements needed to implement HSA [5]



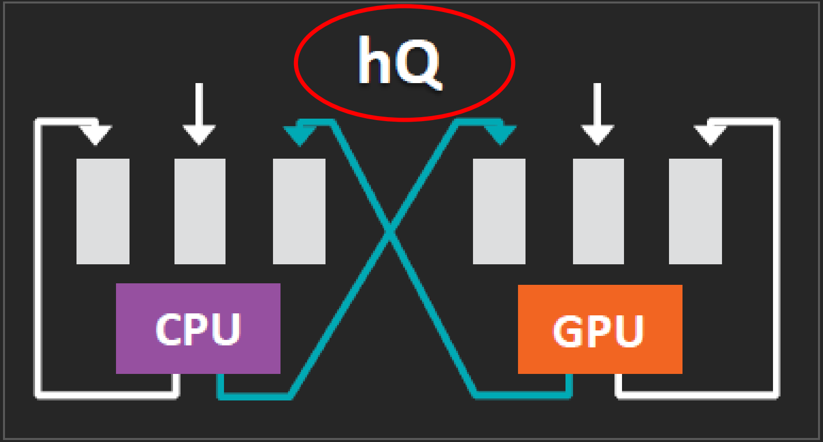
EQUAL ACCESS TO ENTIRE MEMORY



- First time ever: GPU and CPU have uniform visibility into entire memory space (up to 32 GB)

hUMA: heterogeneous UMA
UMA: Uniform Memory Access

EQUAL FLEXIBILITY TO DISPATCH



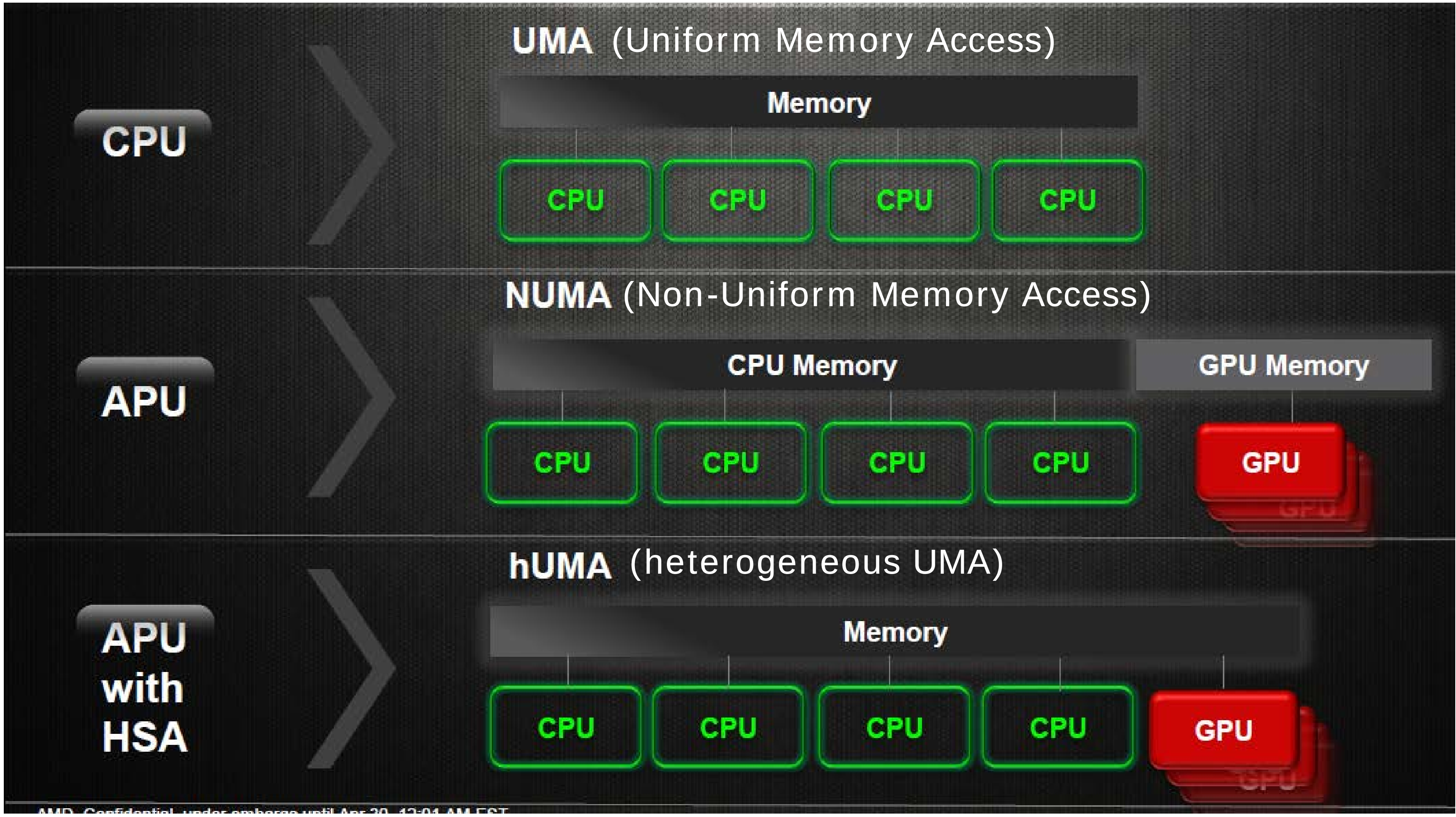
- Heterogeneous queuing (hQ) defines how processors interact equally
- GPU and CPU have equal flexibility to create/dispatch work

2.2 Key hardware enhancements needed to implement HSA (2)

2.2.1 hUMA (Heterogeneous Uniform Memory Access)

2.2 Key hardware enhancements needed to implement HSA (3)

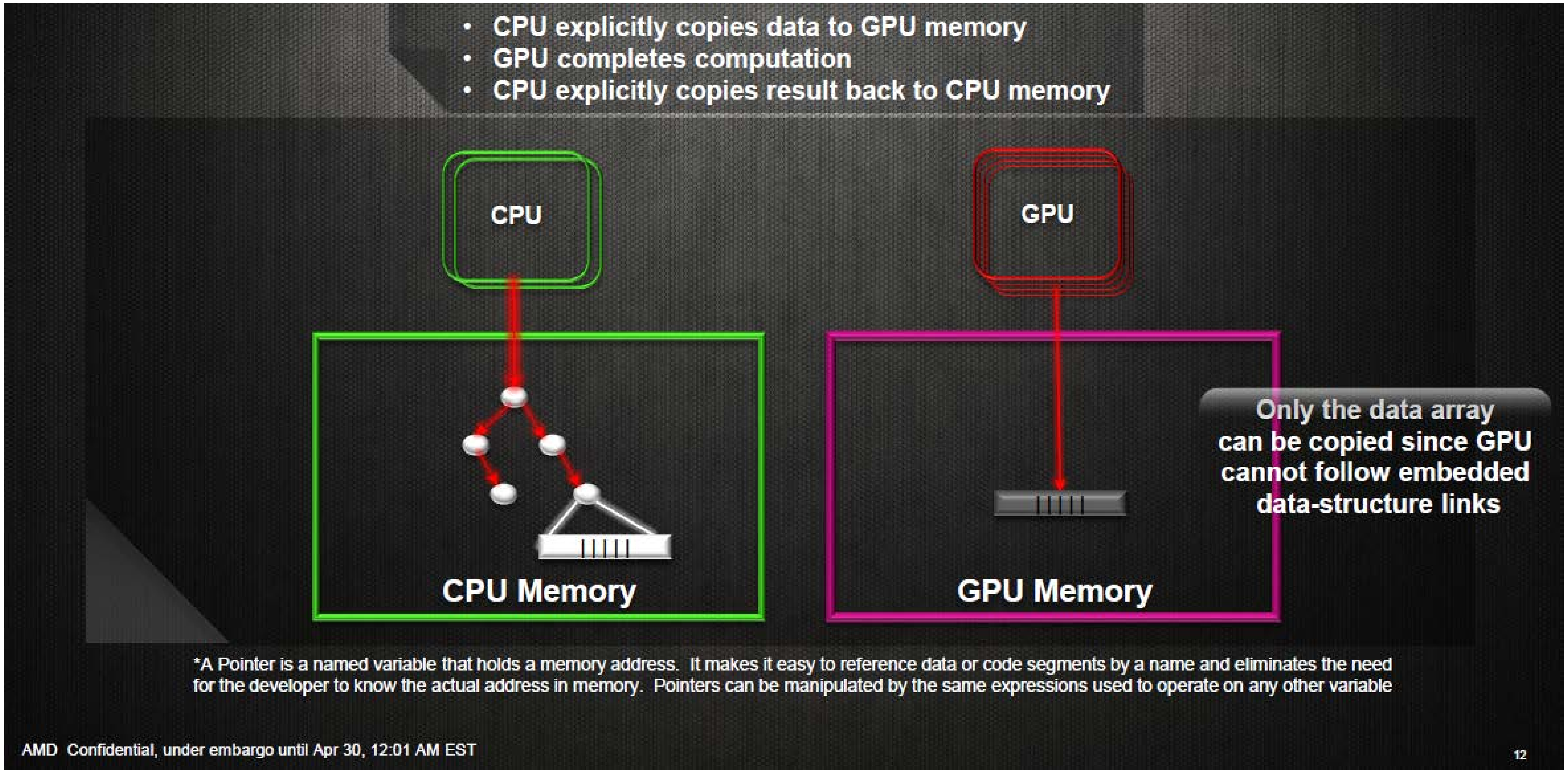
The hUMA (heterogenous UMA) memory access scheme [3]



CPU
UMA (Uniform Memory Access)
Memory
CPU
CPU
CPU
CPU
APU
NUMA (Non-Uniform Memory Access)
CPU Memory
GPU Memory
CPU
CPU
CPU
CPU
GPU
APU with HSA
hUMA (heterogeneous UMA)
Memory
CPU
CPU
CPU
CPU
GPU
AMD Confidential, under embargo until Apr 30, 12:01 AM EST

2.2 Key hardware enhancements needed to implement HSA (4)

GPU co-processing of data structures without hUMA [3]



- CPU explicitly copies data to GPU memory
- GPU completes computation
- CPU explicitly copies result back to CPU memory

CPU
GPU
CPU Memory
GPU Memory
Only the data array can be copied since GPU cannot follow embedded data-structure links

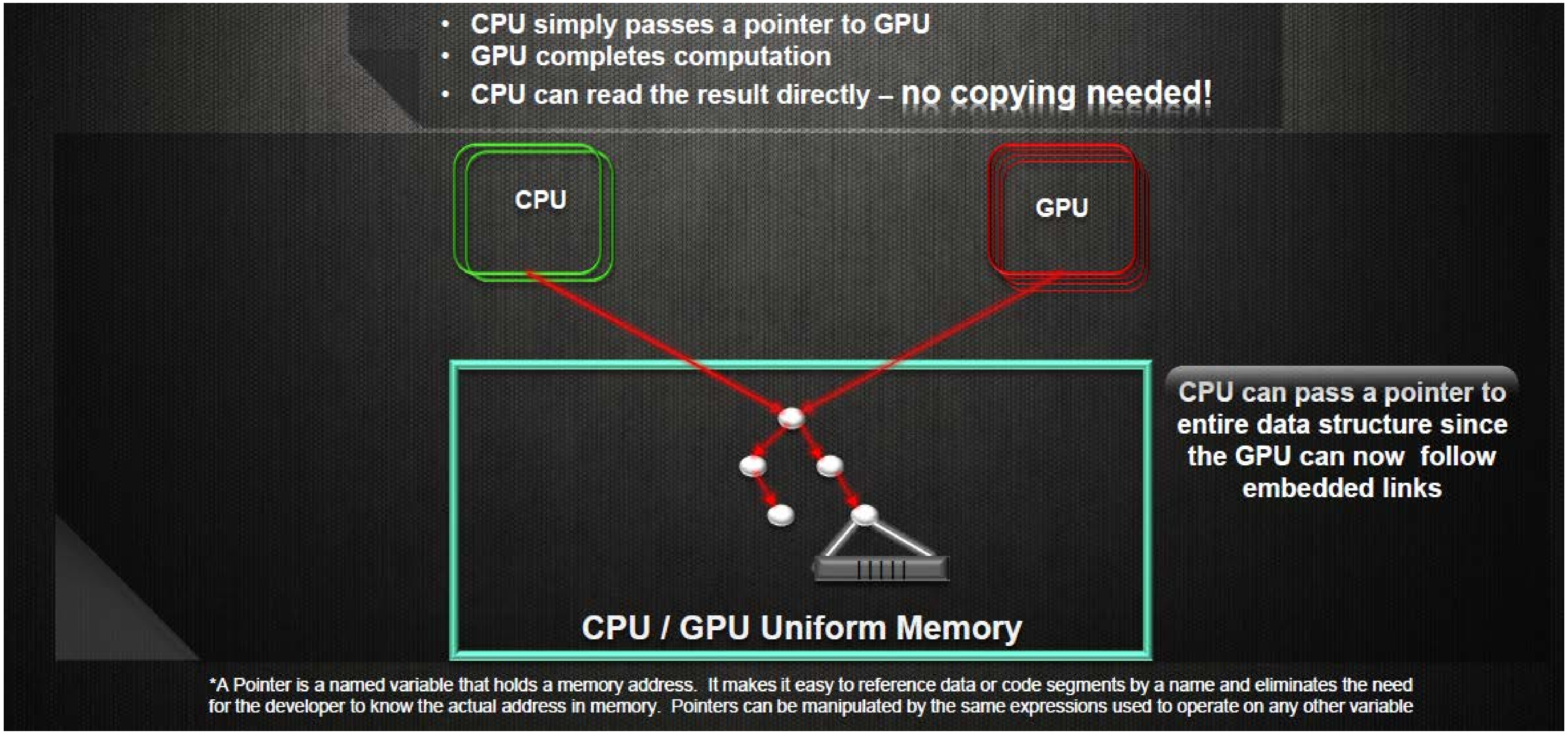
*A Pointer is a named variable that holds a memory address. It makes it easy to reference data or code segments by a name and eliminates the need for the developer to know the actual address in memory. Pointers can be manipulated by the same expressions used to operate on any other variable

AMD Confidential, under embargo until Apr 30, 12:01 AM EST

12

2.2 Key hardware enhancements needed to implement HSA (5)

GPU co-processing of data structures with hUMA [3]



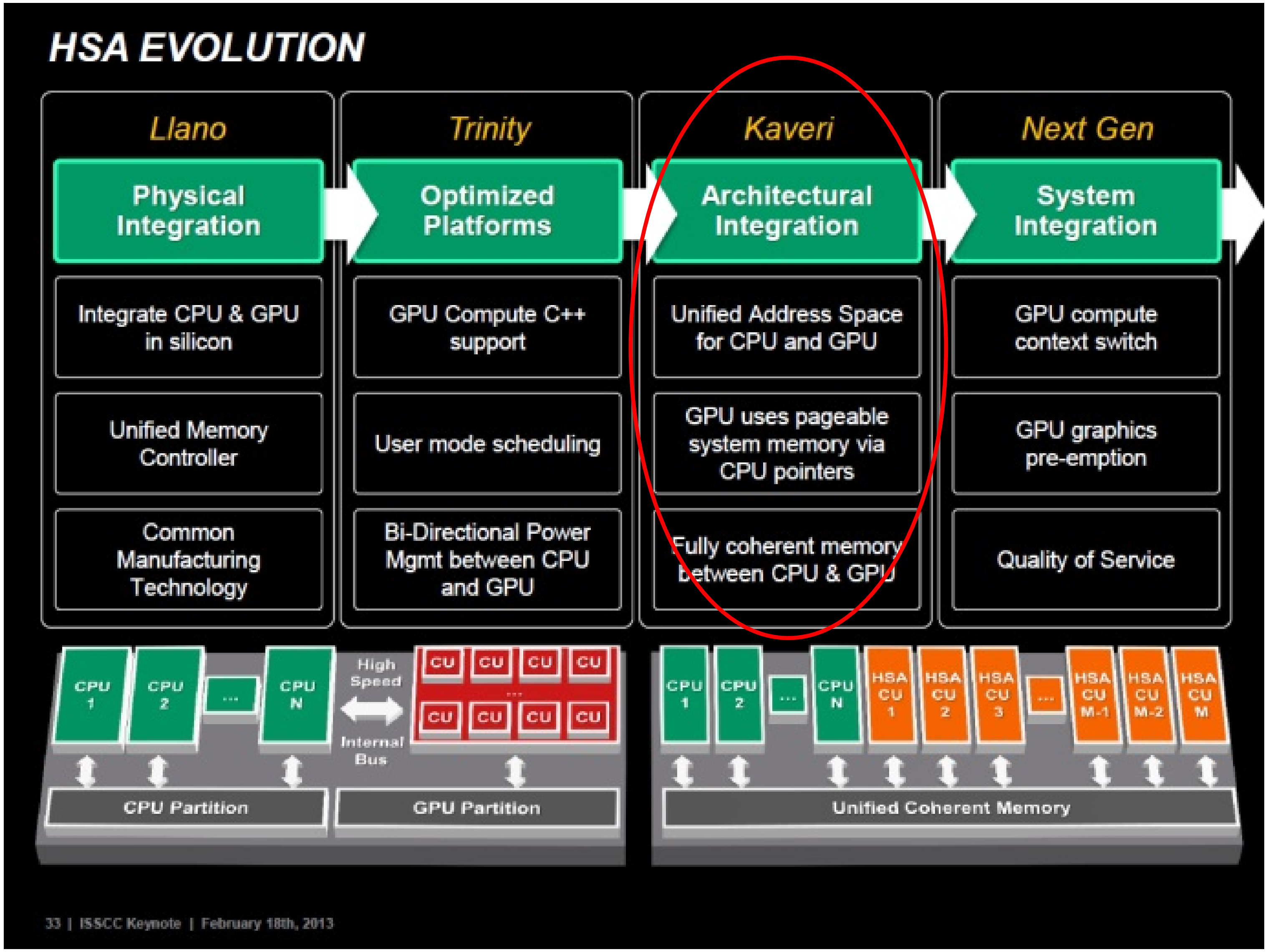
- CPU simply passes a pointer to GPU
- GPU completes computation
- CPU can read the result directly – no copying needed!

CPU can pass a pointer to entire data structure since the GPU can now follow embedded links

*A Pointer is a named variable that holds a memory address. It makes it easy to reference data or code segments by a name and eliminates the need for the developer to know the actual address in memory. Pointers can be manipulated by the same expressions used to operate on any other variable

2.2 Key hardware enhancements needed to implement HSA (6)

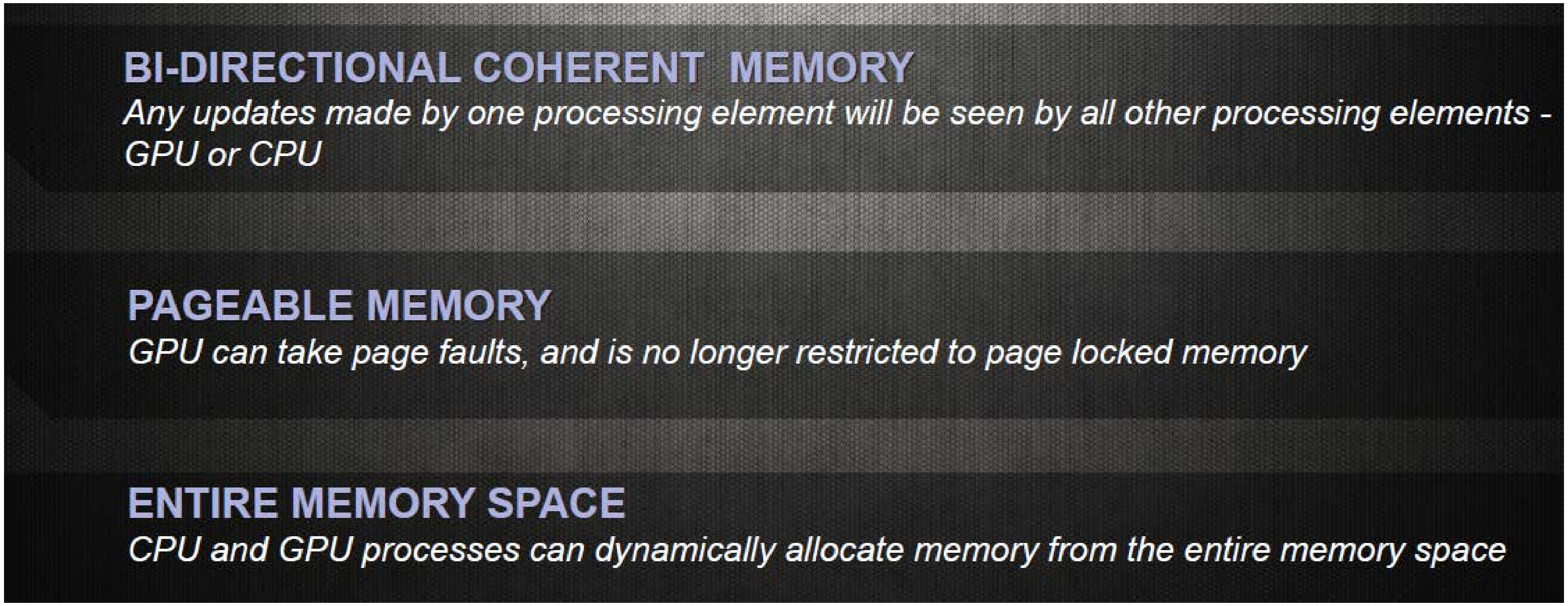
The evolution path to HSA in AMD's subsequent Family 15h based APU lines [7]



HSA EVOLUTION
Llano
Physical Integration
Integrate CPU & GPU in silicon
Unified Memory Controller
Common Manufacturing Technology
Trinity
Optimized Platforms
GPU Compute C++ support
User mode scheduling
Bi-Directional Power Mgmt between CPU and GPU
Kaveri
Architectural Integration
Unified Address Space for CPU and GPU
GPU uses pageable system memory via CPU pointers
Fully coherent memory between CPU & GPU
Next Gen
System Integration
GPU compute context switch
GPU graphics pre-emption
Quality of Service
CPU 1
CPU 2
CPU N
High Speed
Internal Bus
CU
CPU Partition
GPU Partition
HSA CU 1
HSA CU 2
HSA CU 3
HSA CU M-1
HSA CU M-2
HSA CU M
Unified Coherent Memory
33 | ISSCC Keynote | February 18th, 2013

2.2 Key hardware enhancements needed to implement HSA (7)

Key requirements of hUMA-1 [3]



BI-DIRECTIONAL COHERENT MEMORY

Any updates made by one processing element will be seen by all other processing elements - GPU or CPU

PAGEABLE MEMORY

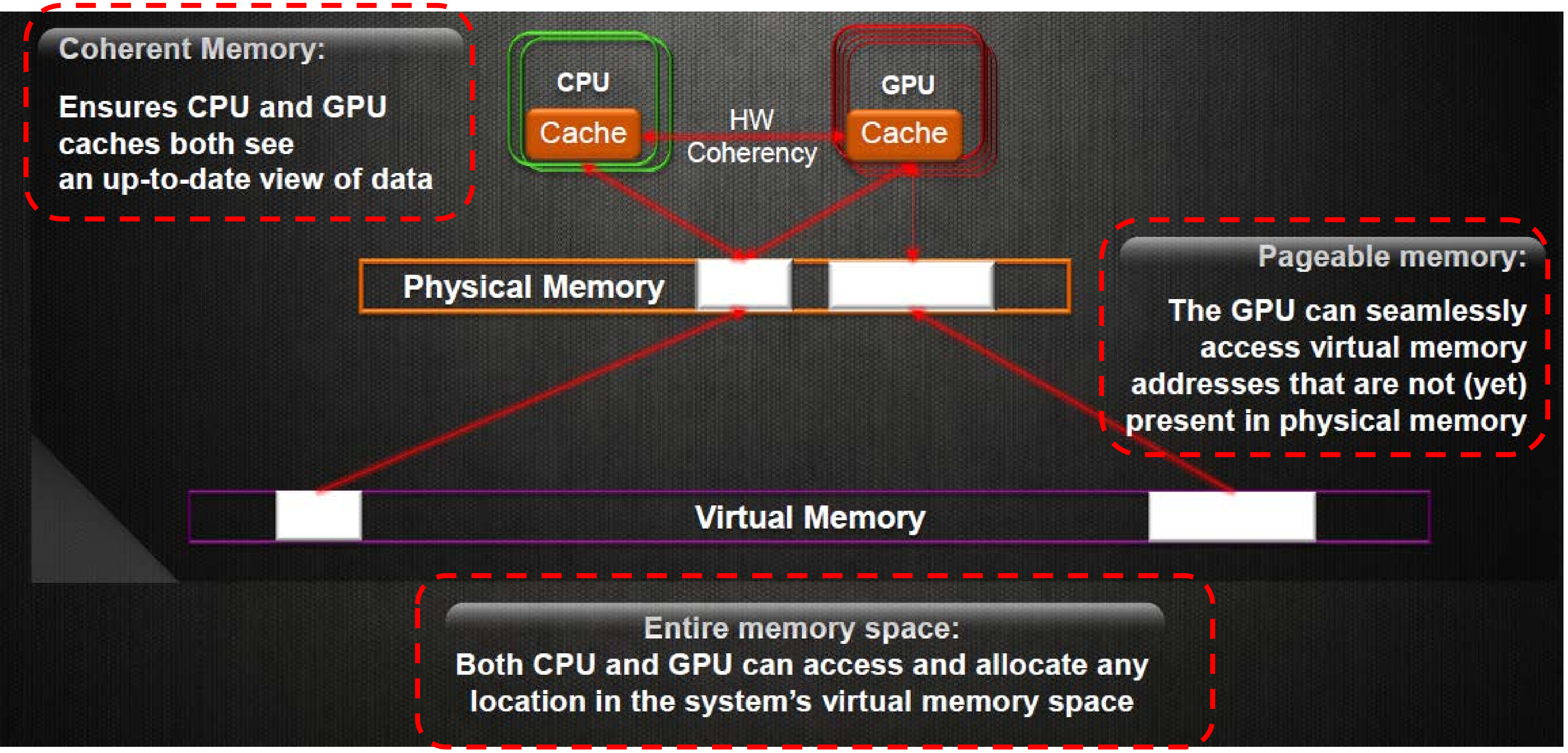
GPU can take page faults, and is no longer restricted to page locked memory

ENTIRE MEMORY SPACE

CPU and GPU processes can dynamically allocate memory from the entire memory space

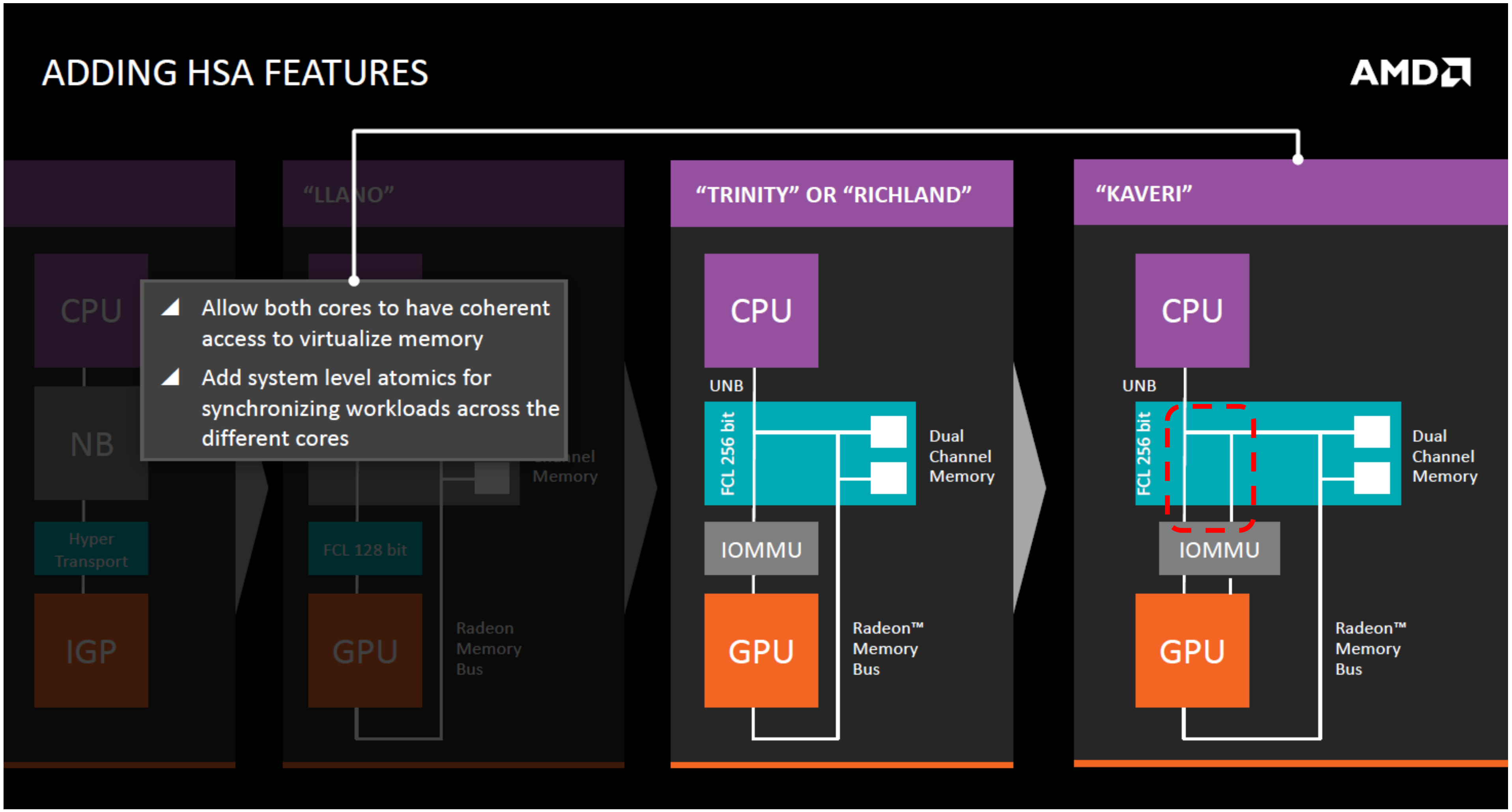
2.2 Key hardware enhancements needed to implement HSA (8)

Key requirements of hUMA-2 [3]


Coherent Memory:
Ensures CPU and GPU caches both see an up-to-date view of data
CPU
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

2.2 Key hardware enhancements needed to implement HSA (9)

Hardware support of memory management in Kaveri [8]



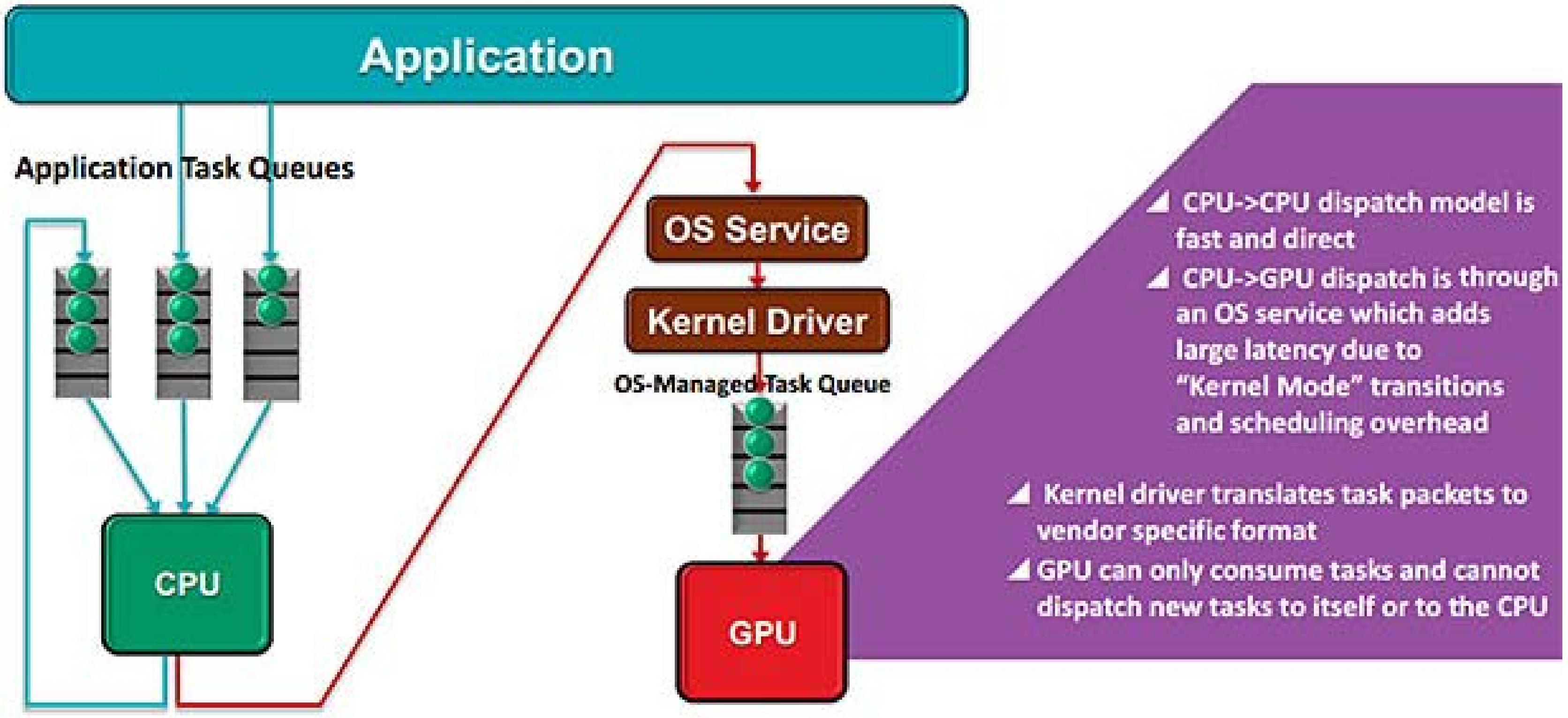
- Allow both cores to have coherent access to virtualize memory
- Add system level atomics for synchronizing workloads across the different cores

2.2 Key hardware enhancements needed to implement HSA (10)

2.2.2 hQ (heterogeneous Queuing)

2.2 Key hardware enhancements needed to implement HSA (11)

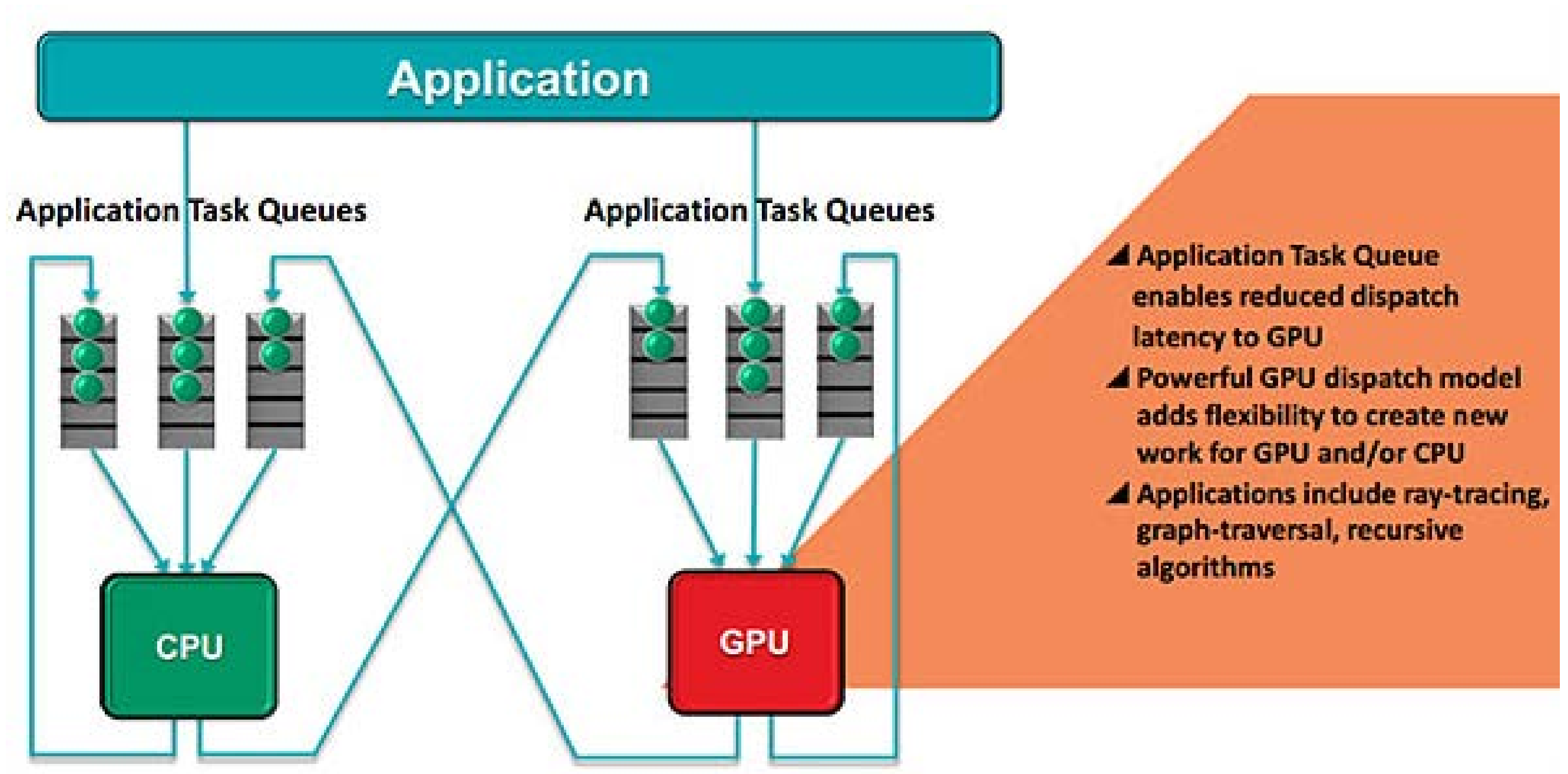
Traditional management of application tasks queues [9]



- CPU->CPU dispatch model is fast and direct
- CPU->GPU dispatch is through an OS service which adds large latency due to "Kernel Mode" transitions and scheduling overhead
- Kernel driver translates task packets to vendor specific format
- GPU can only consume tasks and cannot dispatch new tasks to itself or to the CPU

2.2 Key hardware enhancements needed to implement HSA (12)

Application task management with heterogeneous queuing (hQ) [9]


Application
Application Task Queues
Application Task Queues
CPU
GPU
◢ Application Task Queue enables reduced dispatch latency to GPU
◢ Powerful GPU dispatch model adds flexibility to create new work for GPU and/or CPU
◢ Applications include ray-tracing, graph-traversal, recursive algorithms

2.2 Key hardware enhancements needed to implement HSA (13)

Main features of hQ [9]

- Heterogeneous queuing (hQ) is symmetrical.

  It allows both the CPU and the GPU to generate tasks for themselves and for each other.
- Work is specified in a standard packet format that will be supported by all HSA-compatible hardware, so there's no need for the software to use vendor-specific code.
- Applications can put packets directly into the task queues that will be accessed by the hardware.
- Each application can have multiple task queues, and a virtualization layer allows HSA hardware to see all the queues.

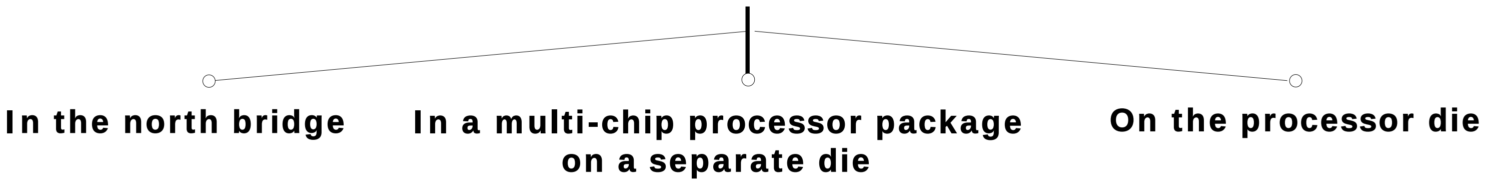
3. Intel's approach to support heterogeneous computing

3.1 Intel's implementations of integrated graphics (1)

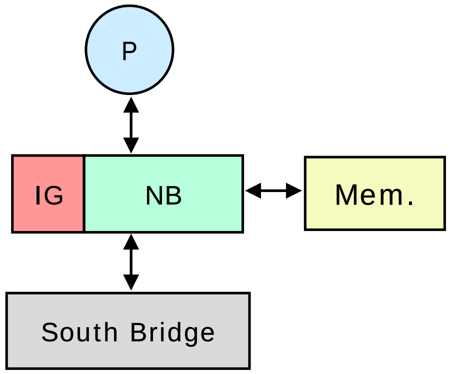
3.1 Intel's implementations of integrated graphics

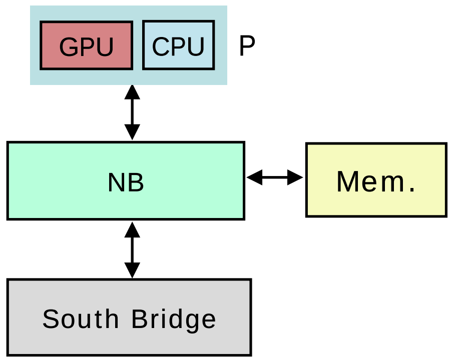
Intel has a long history of integrated graphics, implemented first in the north bridge in the 8xx chipset (1999), as indicated below.

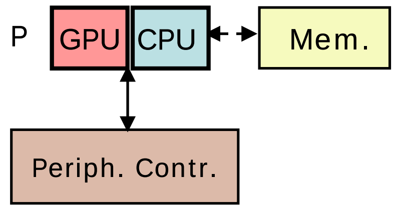
Implementation of integrated graphics



Both the CPU and the GPU are on separate dies and are mounted into a single package







Implementations about
1999 – 2009 such as
Intel's 828xx chipset (1999)
and subsequent implementations

Intel's Havendale (DT) and
Auburndale (M)
(scheduled for 1H/2009
but cancelled)

Arrandale (DT, 1/2010) and
Clarkdale (M, 1/2010)

Intel's Sandy Bridge (1/2011) and
subsequent processors

AMD's Swift (scheduled for 2009
but canceled)

AMD's Bobcat-based APUs (M, 1/2011)
Llano APUs (DT, 6/2011)

and subsequent processors

3.1 Intel's implementations of integrated graphics (2)

Unified Memory Architecture (UMA)

Some early graphics cards of other vendors and essentially all (in the north bridge) integrated graphics designs of Intel already made use of the UMA design as early as in the second half of the 1990s, as the next Figure shows for the integrated graphics of the Intel 810 chipset (1999)

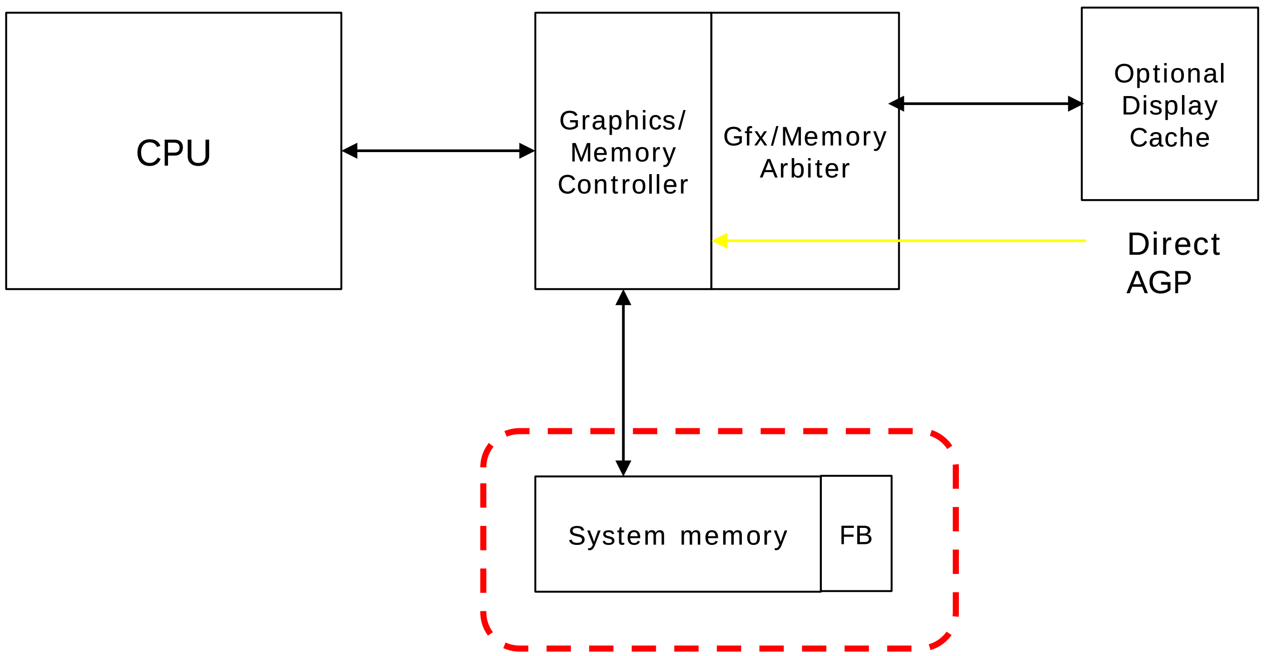


Figure: The UMA design of an early (in the north bridge) integrated graphics [12]

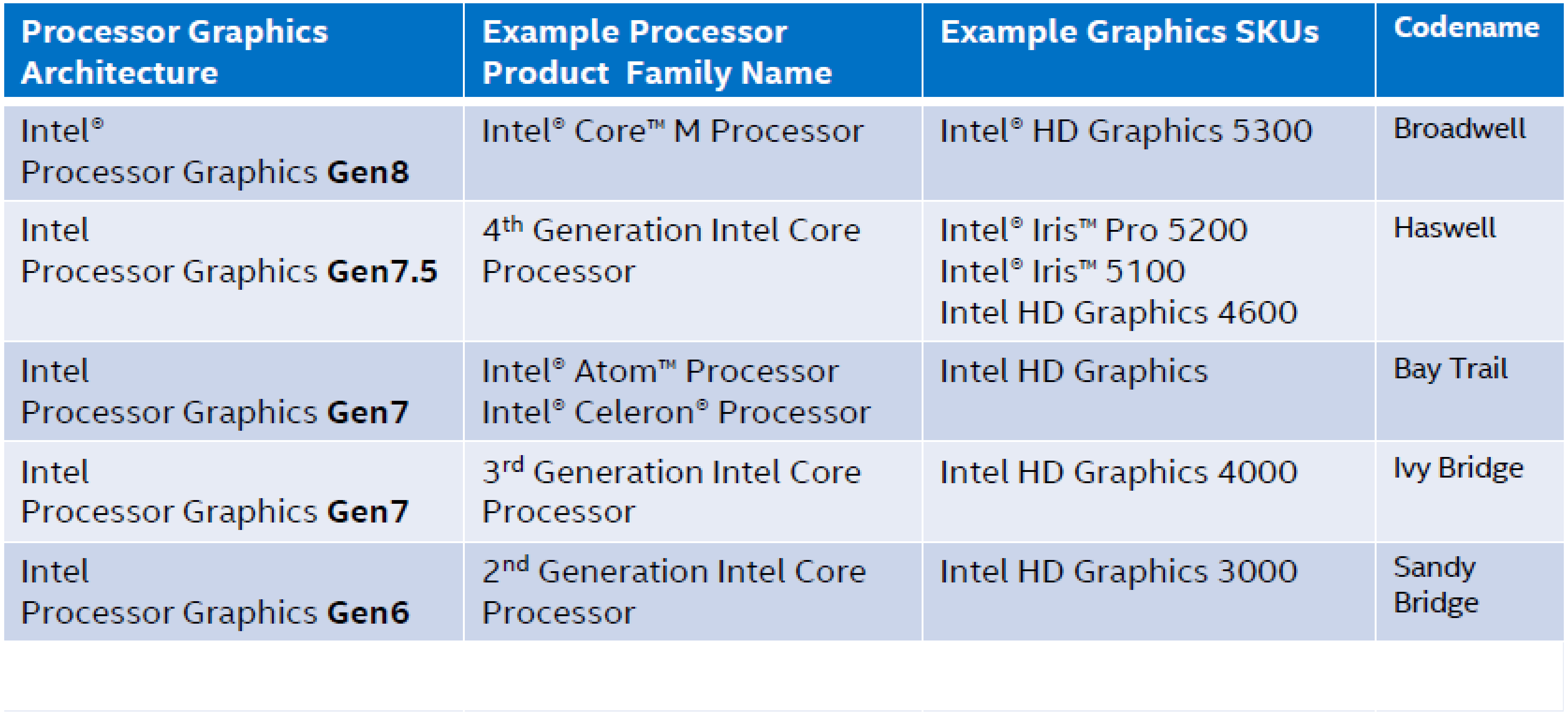
Key benefit/drawback of the UMA design

The UMA design eliminates the need for an extra graphics memory controller and bus but has the constraints of a reduced memory bandwidth.

3.1 Intel's implementations of integrated graphics (3)

Intel's on-die integrated graphics designs

Subsequently, we discuss only those on-die integrated CPU-GPU solutions that support also HPC i.e. have OpenCL support, as follows.



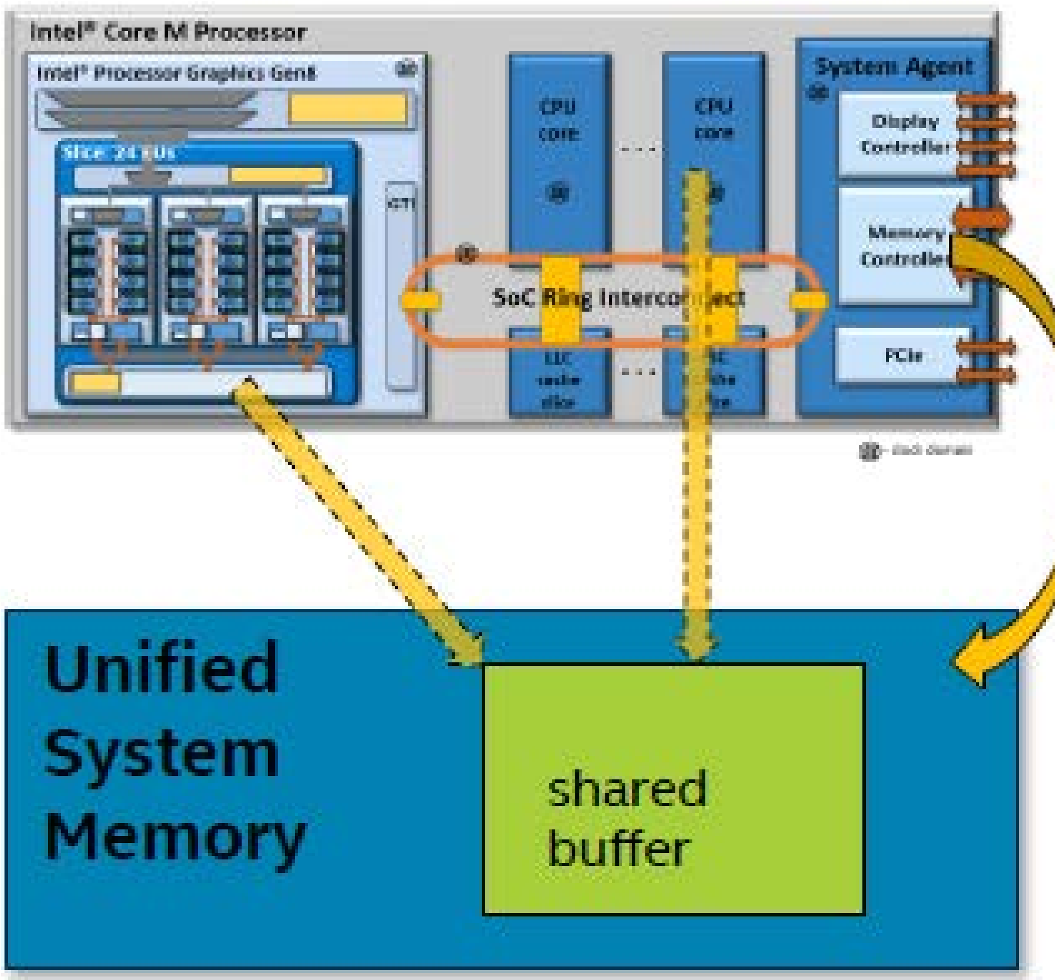
OpenCL support	Processor Graphics Architecture	Example Processor Product Family Name	Example Graphics SKUs	Codename
OpenCL 2.0	Intel® Processor Graphics **Gen8**	Intel® Core™ M Processor	Intel® HD Graphics 5300	Broadwell
OpenCL 1.2	Intel Processor Graphics **Gen7.5**	4th Generation Intel Core Processor	Intel® Iris™ Pro 5200 Intel® Iris™ 5100 Intel HD Graphics 4600	Haswell
OpenCL 1.2	Intel Processor Graphics **Gen7**	Intel® Atom™ Processor Intel® Celeron® Processor	Intel HD Graphics	Bay Trail
OpenCL 1.2	Intel Processor Graphics **Gen7**	3rd Generation Intel Core Processor	Intel HD Graphics 4000	Ivy Bridge
No OpenCL	Intel Processor Graphics **Gen6**	2nd Generation Intel Core Processor	Intel HD Graphics 3000	Sandy Bridge

Table: Intel's processors with on-die integrated GPUs [13]

3.1 Intel's implementations of integrated graphics (4)

Key benefit of OpenCL 1.2 supported Shared Physical Memory, illustrated on an example [13]

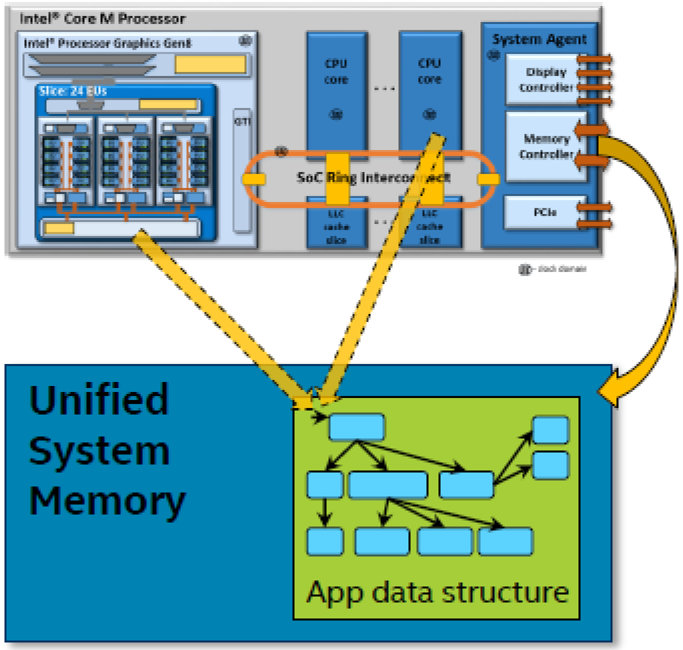
- "Zero Copy" CPU & Graphics data sharing
- Enabled by buffer allocation flags in OpenCL™, DirectX*, etc.



3.1 Intel's implementations of integrated graphics (5)

Shared Virtual Memory - based on the Broadwell architecture and supported by OpenCL 2.0 [13]

- Significant feature, new in Gen8
- Seamless sharing of pointer rich data-structures in a shared virtual address space
- Hardware-supported byte-level CPU & GPU coherency
- Spec'd Intel® VT-d IOMMU features enable heterogeneous virtual memory, page tables, cache snooping protocols…
- Facilitated by OpenCL™ 2.0 Shared Virtual Memory:



3.1 Intel’s implementations of integrated graphics (6)

Key OpenCL 2.0 features [14]

- **Shared Virtual Memory**
  - Host and device kernels can directly share complex, pointer-containing data structures such as trees and linked lists, providing significant programming flexibility and eliminating costly data transfers between host and devices
- **Dynamic Parallelism**
  - Device kernels can enqueue kernels to the same device with no host interaction, enabling flexible work scheduling paradigms and avoiding the need to transfer execution control and data between the device and host, often significantly offloading host processor bottlenecks

3.2 Intel’s first implementation of Shared Virtual Memory
The Core M processor

3.2 Intel’s first implementation of Shared Virtual Memory – The Core M (1)

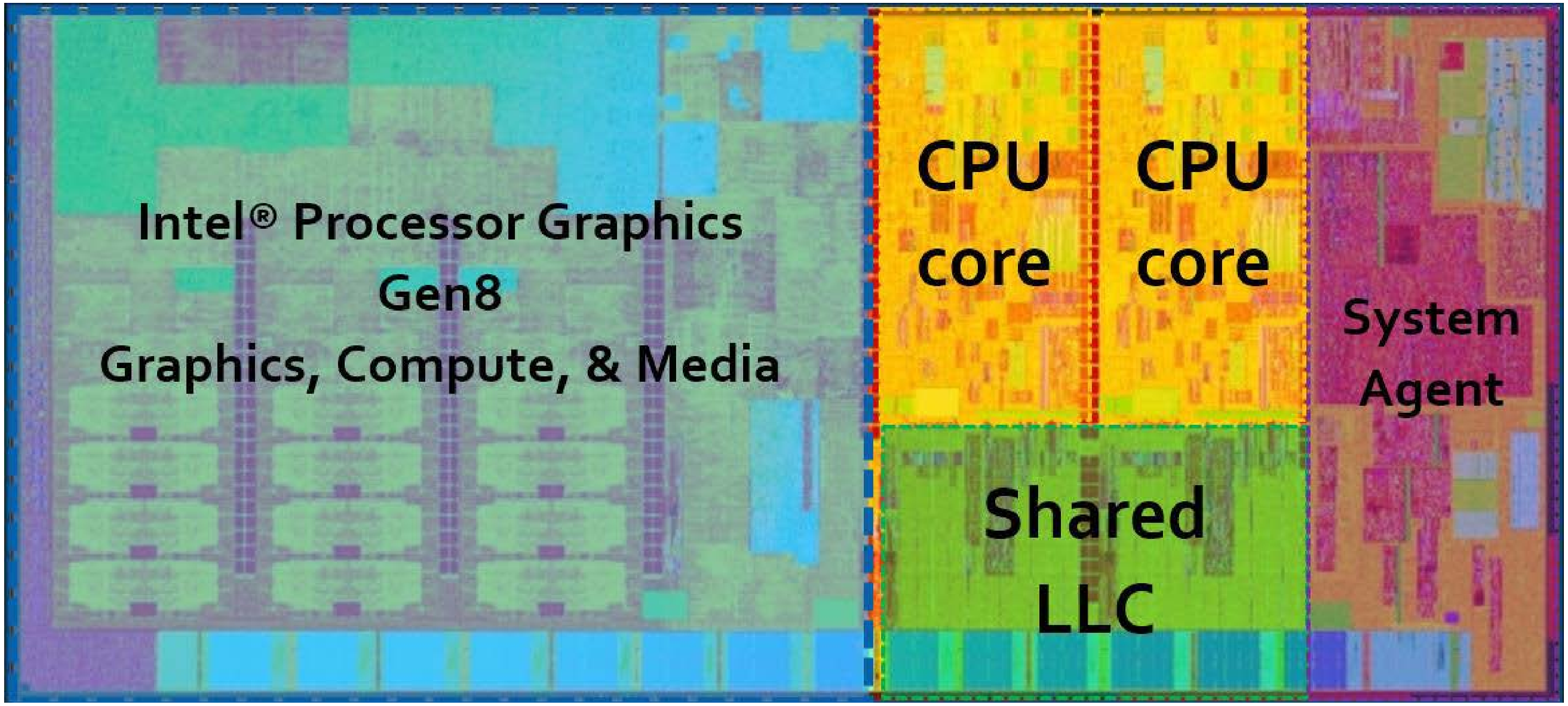
3.2 Intel’s first implementation of Shared Virtual Memory – The Core M processor

- The Core M processor targets tablets and 2-in-1 devices
- Based on the 14 nm Broadwell architecture
- Includes Gen8 graphics
- It is a SOC (System on Chip) design
- Announced: 8/2014

.

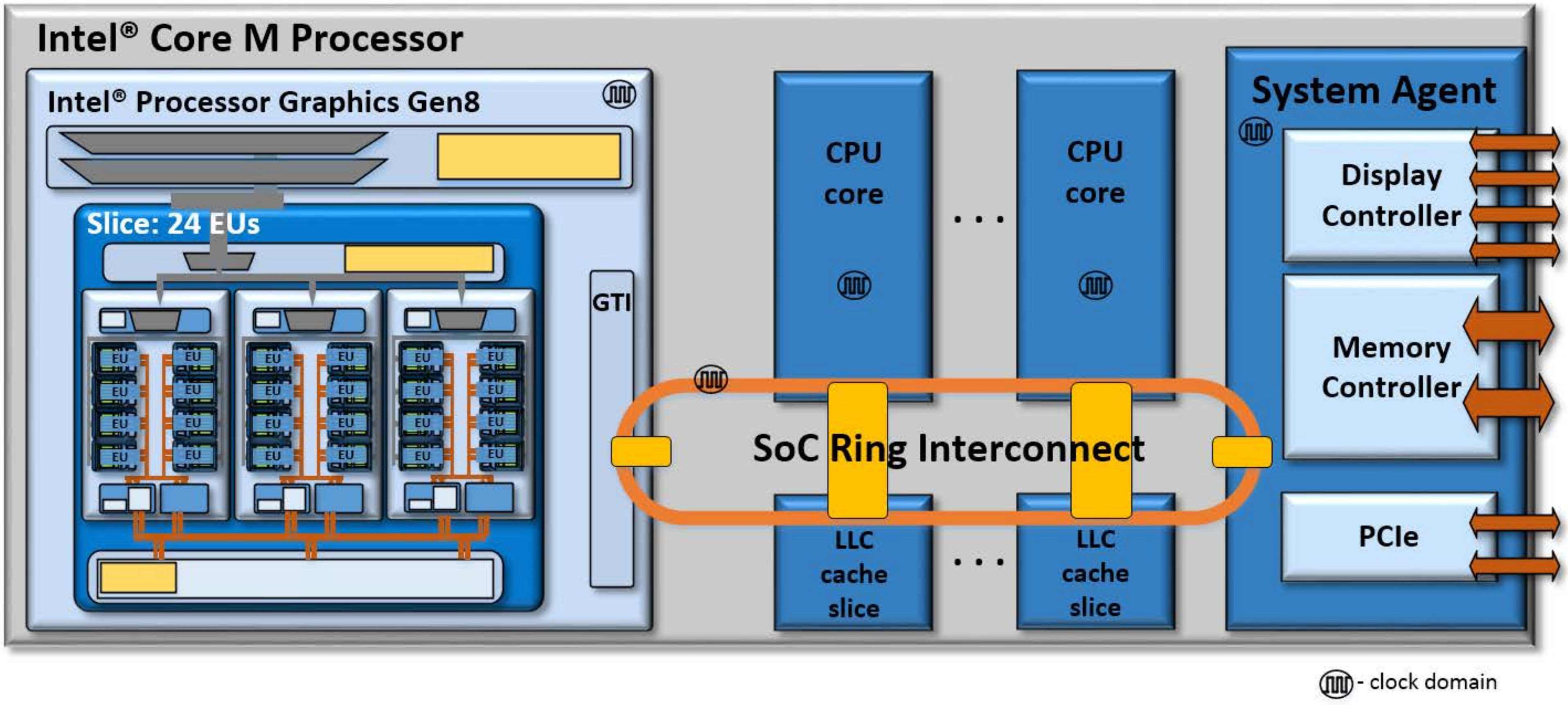
3.2 Intel's first implementation of Shared Virtual Memory – The Core M (3)

Die layout of the Core M [13]



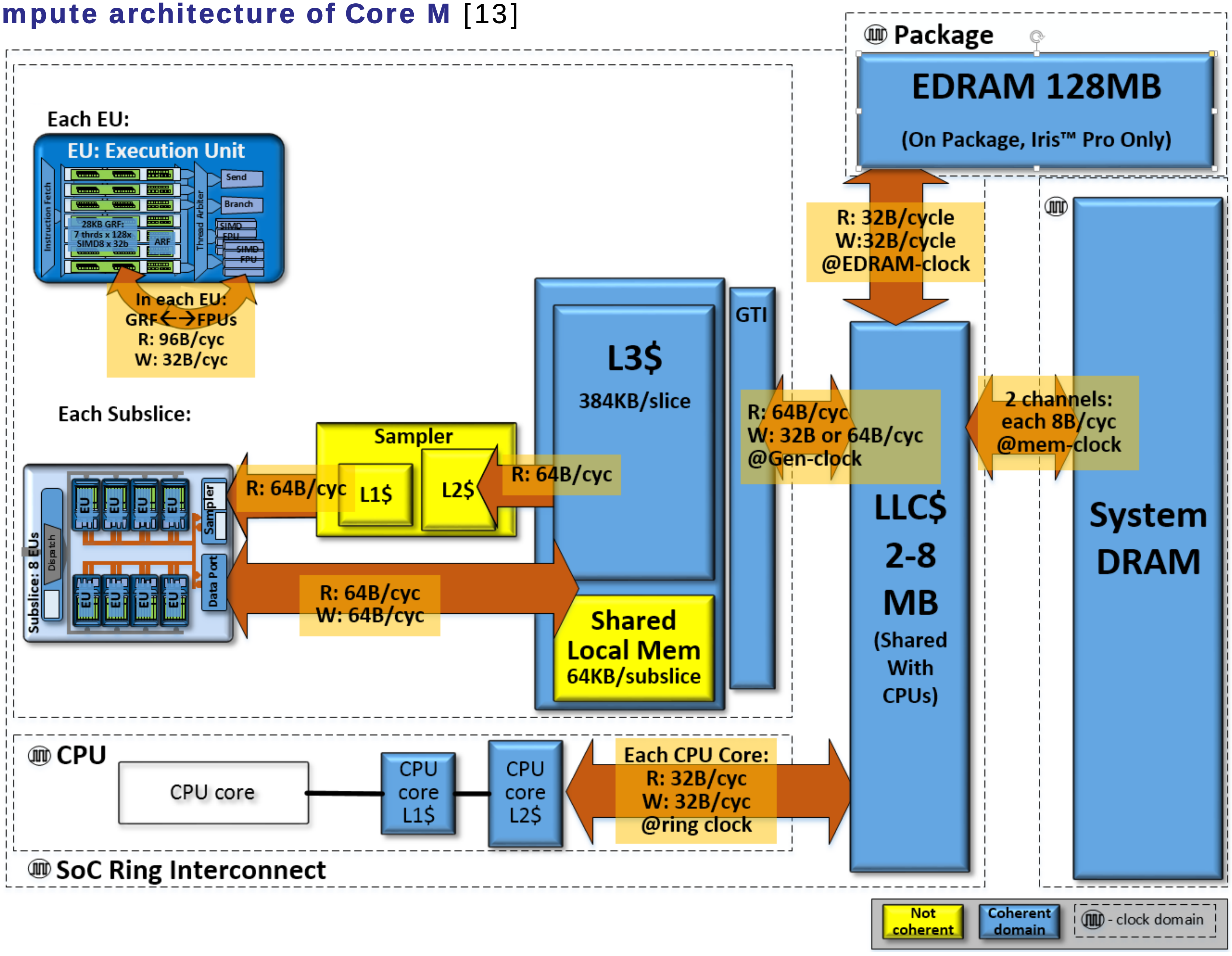
3.2 Intel's first implementation of Shared Virtual Memory – The Core M (4)

Block diagram of the Core M [13]



3.2 Intel's first implementation of Shared Virtual Memory – The Core M (7)

Compute architecture of Core M [13]



Package
EDRAM 128MB
(On Package, Iris™ Pro Only)
R: 32B/cycle
W:32B/cycle
@EDRAM-clock
Each EU:
EU: Execution Unit
Instruction Fetch
Thread Arbiter
28KB GRF: 7 thrds x 128x SIMD8 x 32b
ARF
Send
Branch
SIMD FPU
SIMD FPU
In each EU:
GRF ←→ FPUs
R: 96B/cyc
W: 32B/cyc
Each Subslice:
Subslice: 8 EUs
Dispatch
EU
Sampler
Data Port
Sampler
L1$
L2$
R: 64B/cyc
R: 64B/cyc
R: 64B/cyc
W: 64B/cyc
L3$
384KB/slice
Shared Local Mem
64KB/subslice
GTI
R: 64B/cyc
W: 32B or 64B/cyc
@Gen-clock
LLC$
2-8 MB
(Shared With CPUs)
2 channels:
each 8B/cyc
@mem-clock
System DRAM
CPU
CPU core
CPU core L1$
CPU core L2$
Each CPU Core:
R: 32B/cyc
W: 32B/cyc
@ring clock
SoC Ring Interconnect
Not coherent
Coherent domain
- clock domain