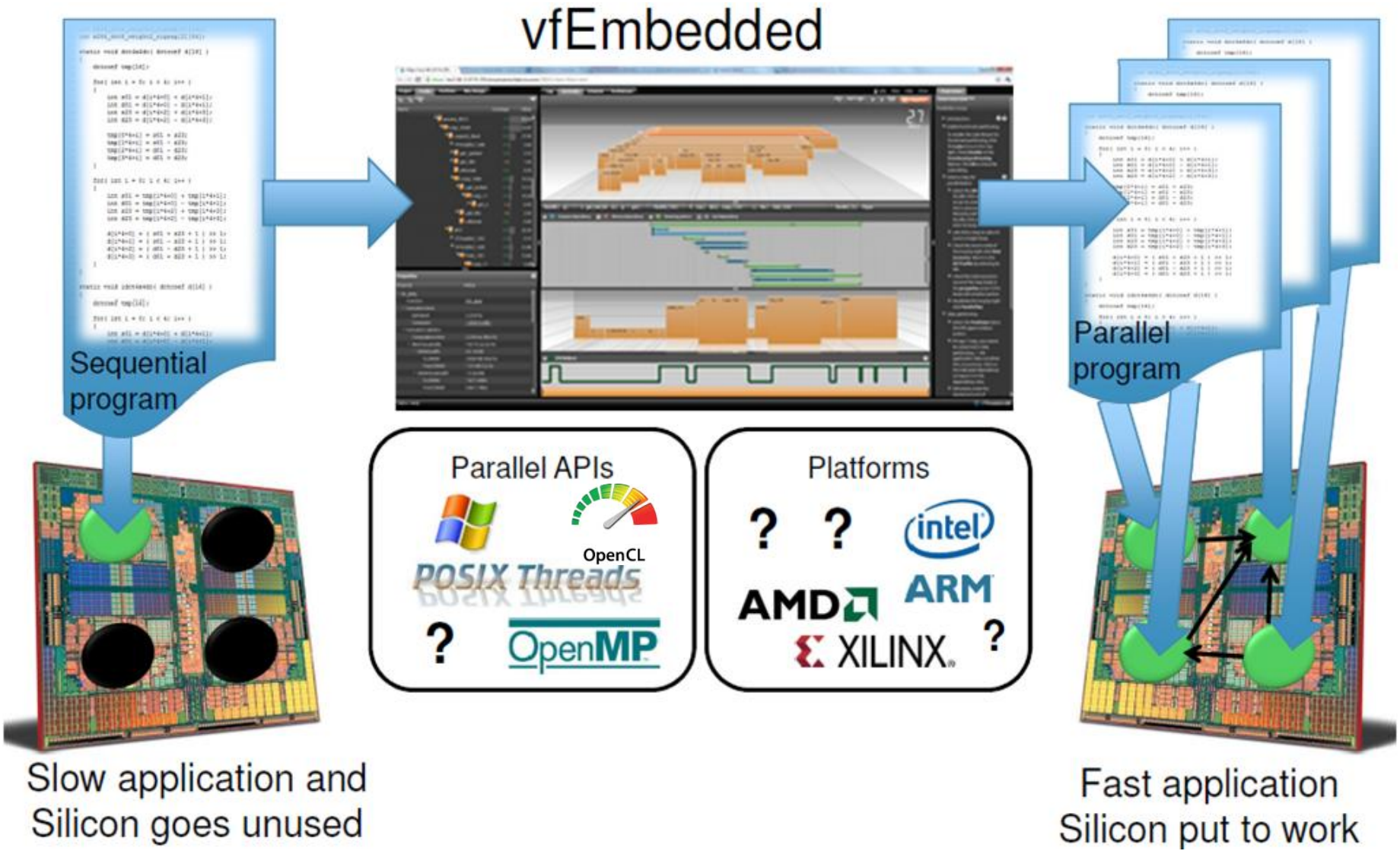
Mapping applications into MPSoC
concurrency & communication

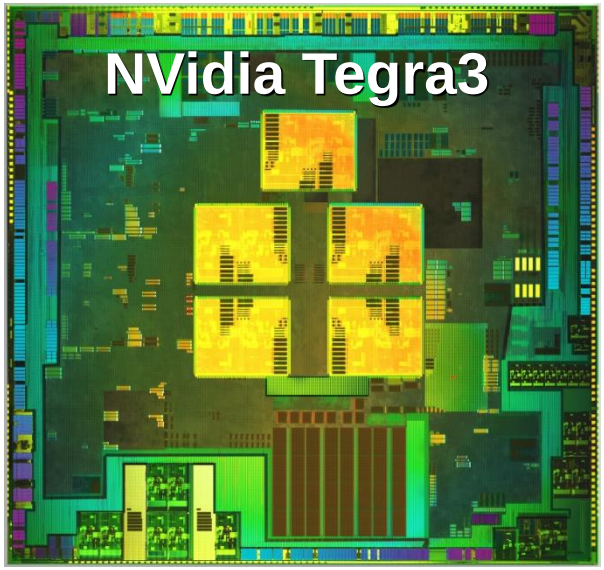
Jos van Eijndhoven
jos@vectorfabrics.com

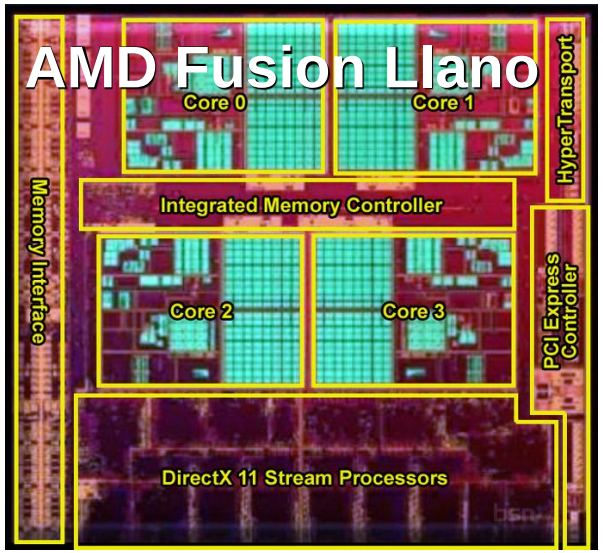
March 12, 2011

MPSoC mapping: exploiting concurrency



Computation on general purpose CPUs





- CPUs are generic work horses
- From 1, to 2, to 4, to ... CPUs to grow performance
- Shared memory abstraction is costly in HW. (multi-level caches, snooping)
- This abstraction is needed for multi-threaded software: applications and operating system.

Area(ALU)/Area(CPU) is very low (1%?)

Computational efficiency beyond CPU

- General-purpose CPUs are (traditionally) designed to handle code with complex control-flow
- DSPs emphasize efficient processing of regular compute

But DSP and CPU architectures are evolving towards each other.

How to significantly increase operations/sec/$ and operations/J ?

Hand-off compute load to:

- Function-specific accelerators
  (H264 decode, LTE channel decode, GFX rendering, IP packet processing, ...)
- GP-GPU: general-purpose programmable graphics processor units
- FPGA accellerators: Field programmable gate arrays

GP-GPU and FPGA allow new workloads on off-the-shelf silicon

MPSoC programmability??

- Silicon technology: incomprehensible growth transistors/chip
- SoC architecture: complex systems, many homogeneous and heterogeneous processor cores, non-uniform memory

Who is the poor programmer to put these devices into good use?

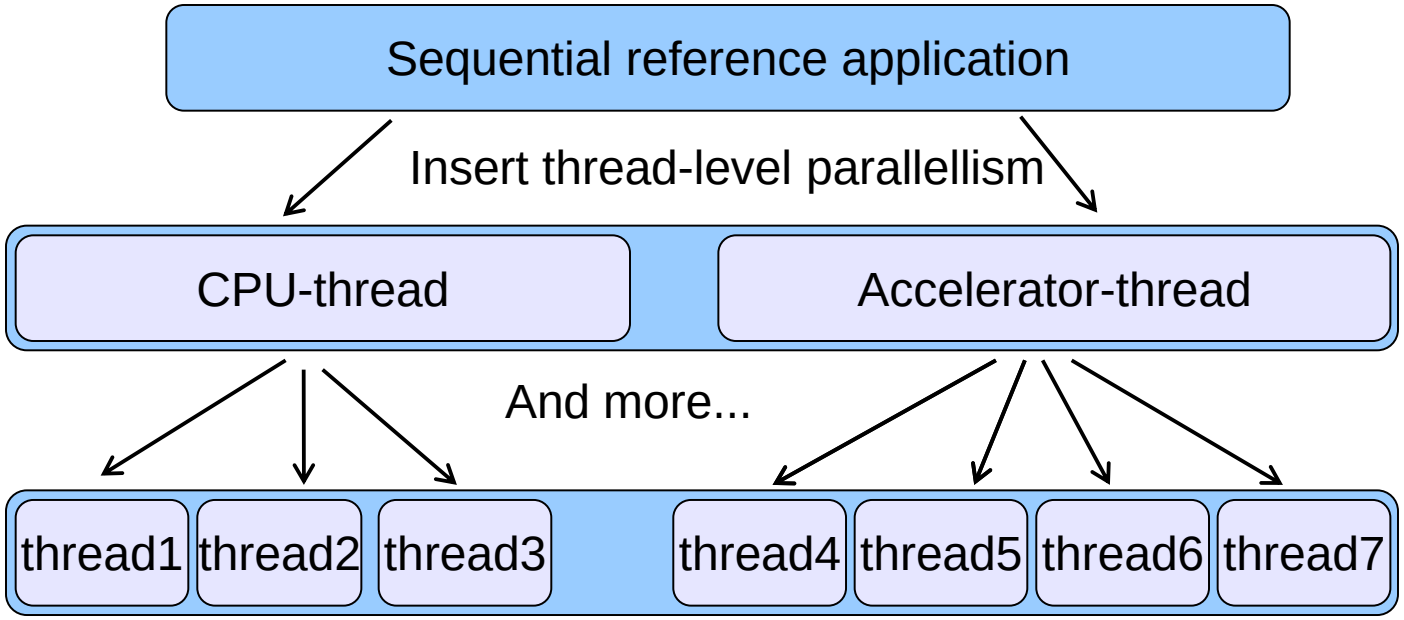
- Limited SW creation cost (re-use & portability)
- Efficient use of HW resources
- Clean SW design (free of bugs)
- Timely delivery
- . . .

Presentation theme: concurrency & data

Programmability of multi-processors:

- Concurrency: distribution of operations
- Memory mapping: distribution of data

Embedded system Application mapping



Create function parallelism

Create data parallelism

Tough issues are not in OPERATIONS, but in DATA:

- Parallelism is typically hindered by data dependencies
- Data must be available in local/nearby memories

Presentation further Sections 1 ... 4

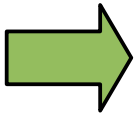
	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	Section 2
System implementation view	Section 3	Section 4

Section 1: application functional pipelining

	Functional pipeline partitioning	Data parallel partitioning
Software Application view	**Section 1**	Section 2
System implementation view	Section 3	Section 4

Function pipelining: partitioning

```
int A[N][M];

while (..)
{ produce_img();
  consume_img();
}

produce_img()
{ for (i ...)
   for (j ...)
    A[i][j] = ...
}

consume_img()
{ for (i ...)
   for (j ...)
     ... = A[i][j];
}
```



Loop distribution:

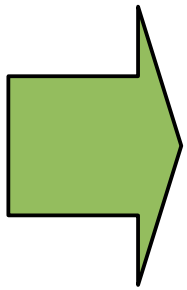
```
Thread1: while (..)
   produce_img();

Thread2: while (..)
   consume_img();
```

Synchronize thread progress:

- True dependency: consumer must wait for valid data
- Anti dependency: producer must wait with over-writing until after consumption

Function pipelining: synchronization

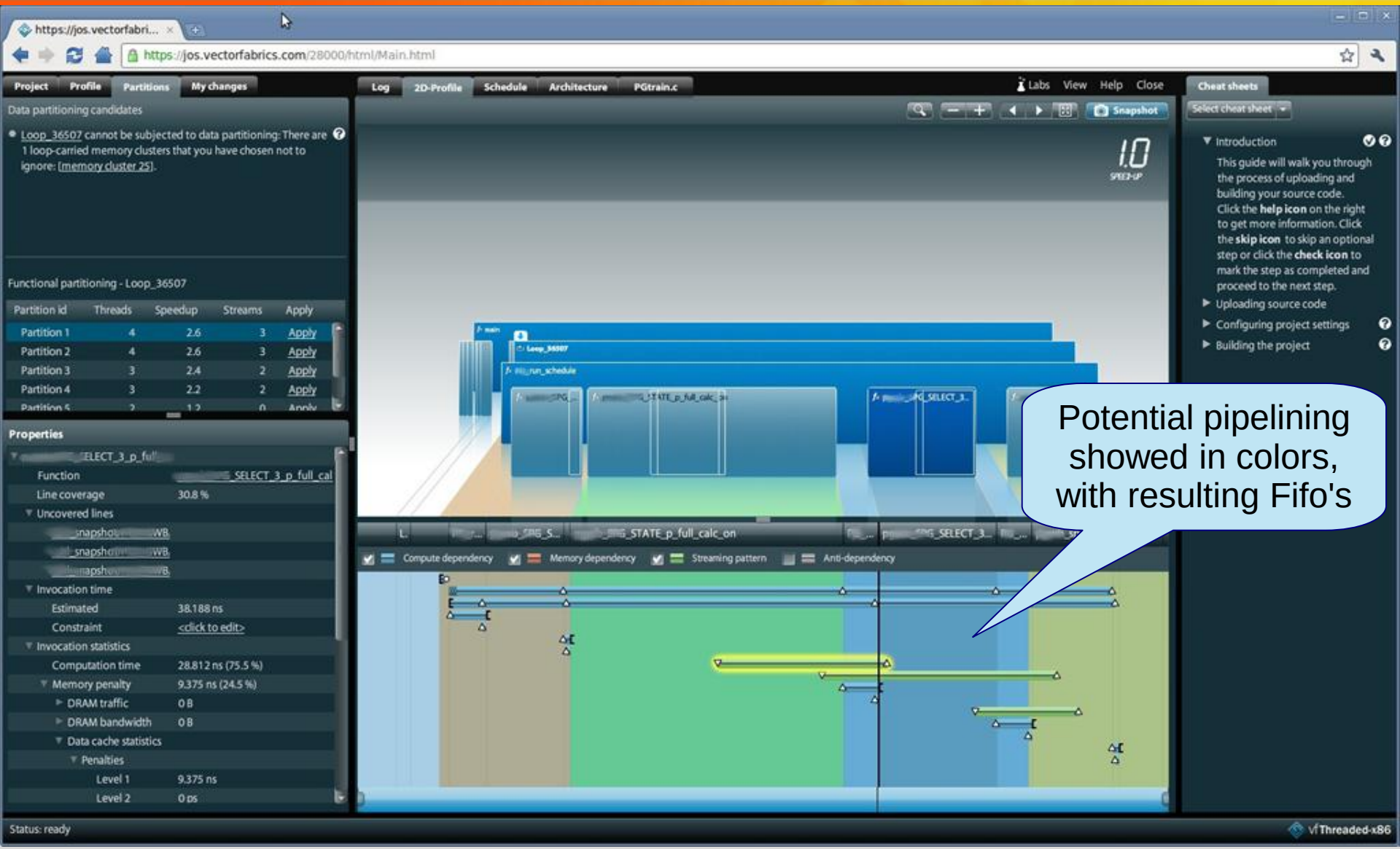
```
int A[N][M];

while (..)
{ produce_img();
  consume_img();
}

produce_img()
{ for (i ...)
    for (j ...)
     A[i][j] = ...
}

consume_img()
{ for (i ...)
    for (j ...)
      ... = A[i][j];
}
```



```
Channel ch;

Thread1: while (..)
   produce_img();

Thread2: while (..)
   consume_img();

produce_img()
{ for (i ...)
    for (j ...)
     write_int( ch, ...)
}

consume_img()
{ for (i ...)
    for (j ...)
      ... = read_int(ch);
}
```

Channel access functions implement thread stall.

Pipeline dependency analysis



Function pipelining: Channel APIs

Too many choices for channel-based communication:

- Standard Java util.concurrent queue classes
- Intel's TBB (C++) queues
- Linux 'pipes' and 'sockets'
- OpenCL channels
- OpenMAX IL for streaming media processing
- MPI message-passing channels
- . . .

Very different queue implementations:

- Inter-thread, inside process memory context
- Inter-process, inside shared-memory system
- Inter-system, through device interfaces

NOTE: C++ STL queues are NOT thread-safe!

Section 2: data parallelism in applications

	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	**Section 2**
System implementation view	Section 3	Section 4

Data parallelization: multi-core scalability

```
int sum = 0;
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Distribute the workload over multiple cores.
- Each core handles part of the loop index space.

```
int sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Workload scales nicely across multiple cores
- Easy to write down ☺, but hard to grasp all consequences!
- Highly dangerous, might cause extremely hard-to-track bugs! ☹

Application Analysis

Section 3: coprocessors and data channels

	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	Section 2
System implementation view	**Section 3**	Section 4

GPU or FPGA next to CPU

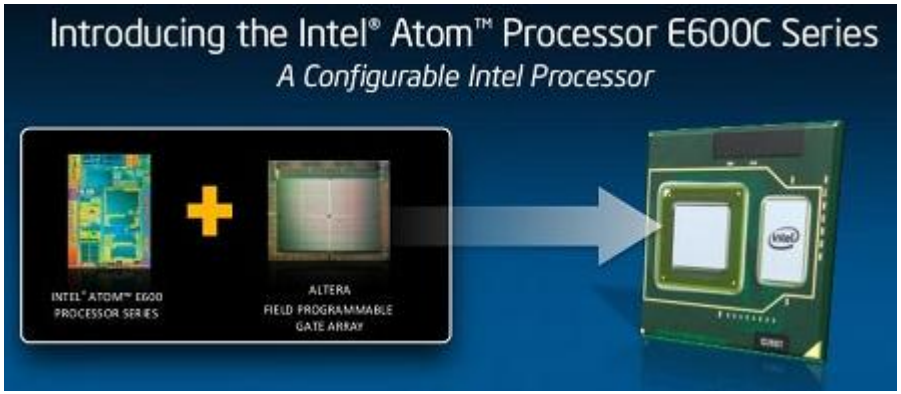
GP-GPU:

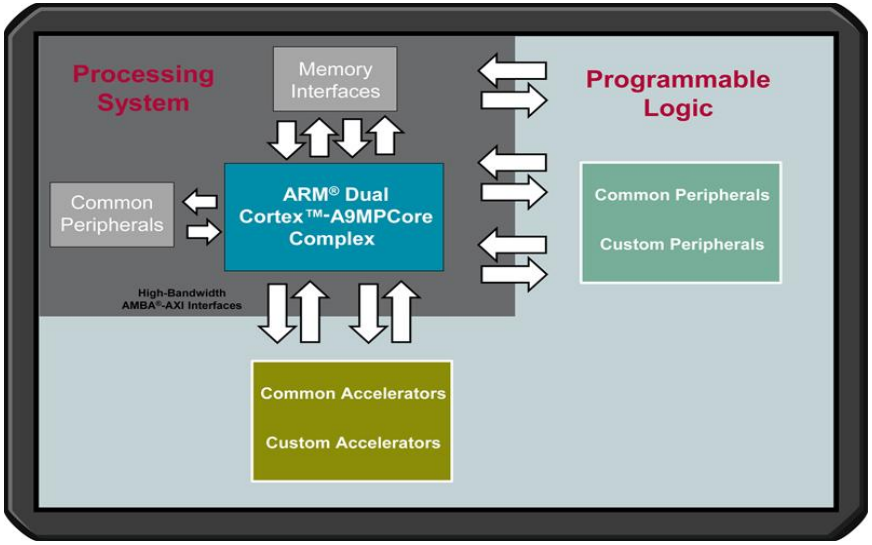
- High floating point performance (>1TFlops)
- Large off-chip memory bandwidth
- Needs thousands of concurrent threads (SPMD)
- Few inter-thread data dependencies and little data-dependent control
- High-end chips take huge power (>100W)

FPGA:

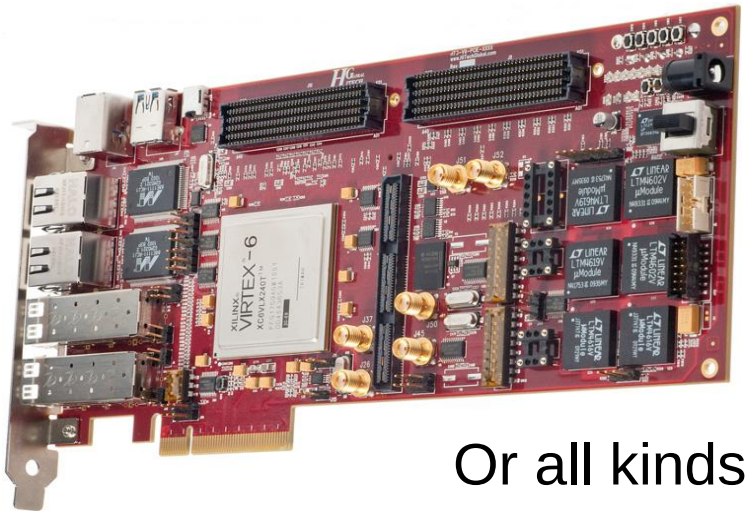
- High integer performance (>1Tops)
- Application-specific off-chip data interfaces.
- Needs hundreds of concurrent instructions (ILP)
- Takes HW design expertise and effort.
- High-end chips are very expensive (>$1000)

CPU – FPGA combinations



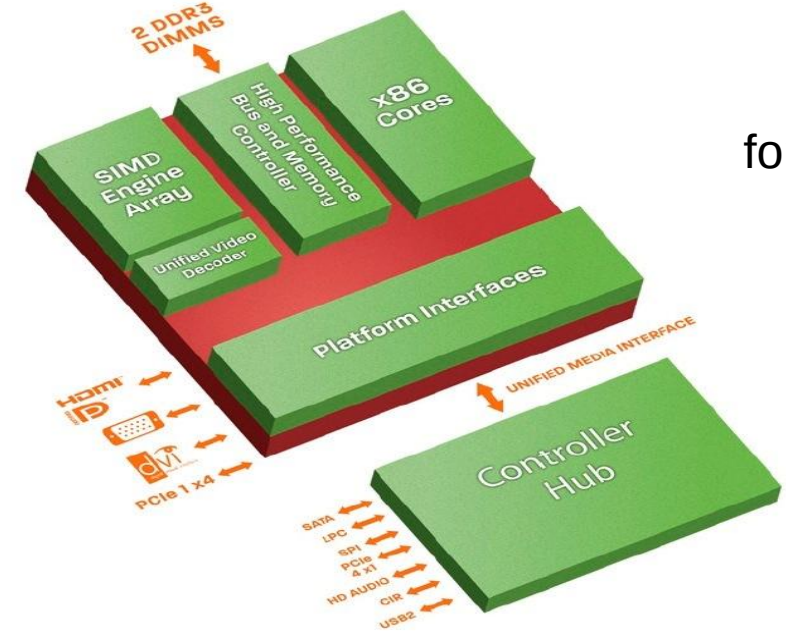


Xilinx 'Zync' contains dual ARM



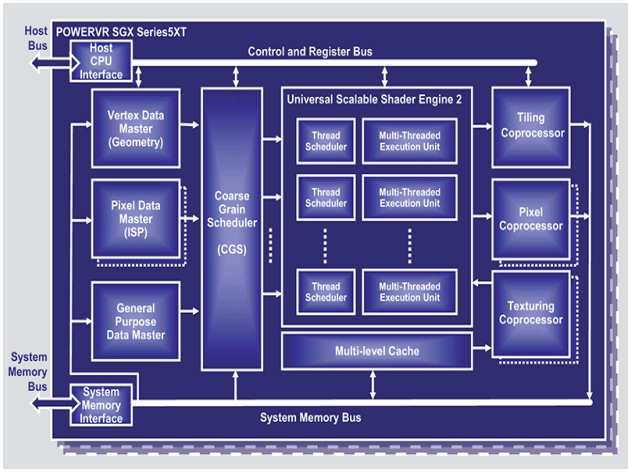
Or all kinds of boards to fit PC architecture

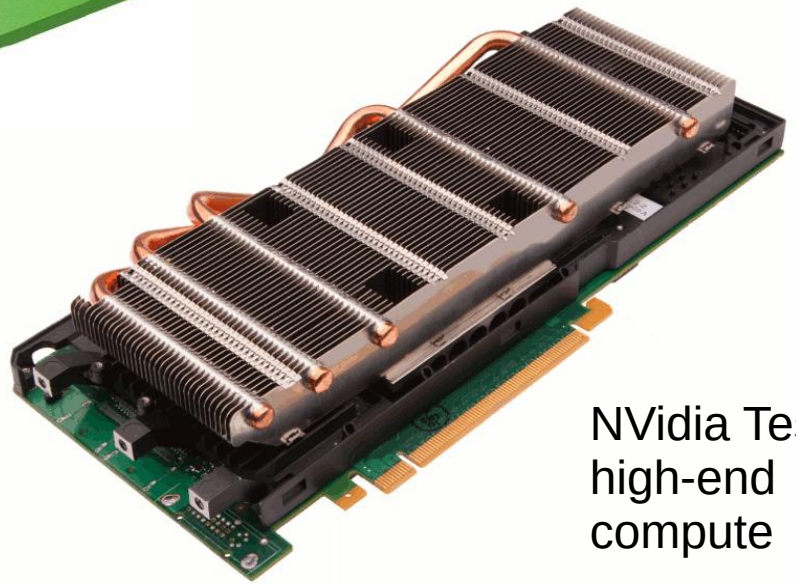
CPU – GPGPU combinations



AMD Fusion for Tablet, Desktop, ...

Imgtec PowerVR for mobile SoC

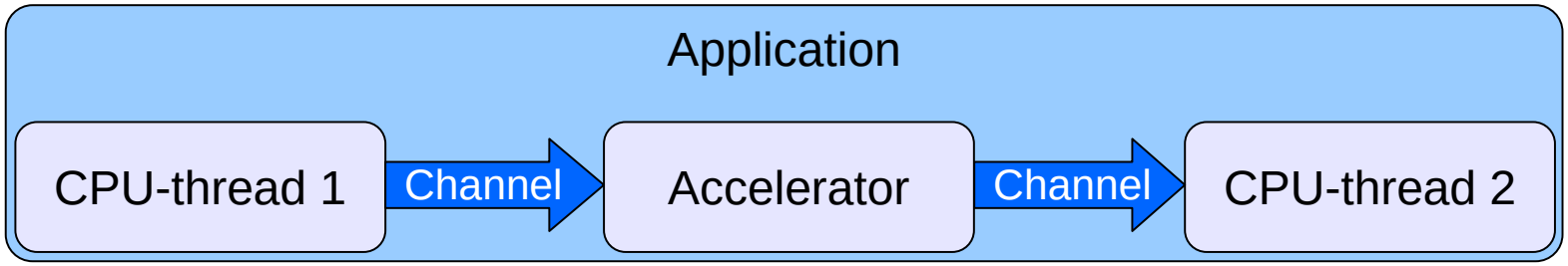




NVidia Tesla for high-end compute

Application - Accelerator co-processing

Functional pipelining, some thread mapped to HW accelerator, *channels* for inter-thread data transport:



Conceptually nice picture, real implementation hurdles:

- Application I/O to hardware is shielded by any 'real' operating system
- Thread control (sleep/wakeup) interacts with Accelerator progress

Memory-mapped channel communication

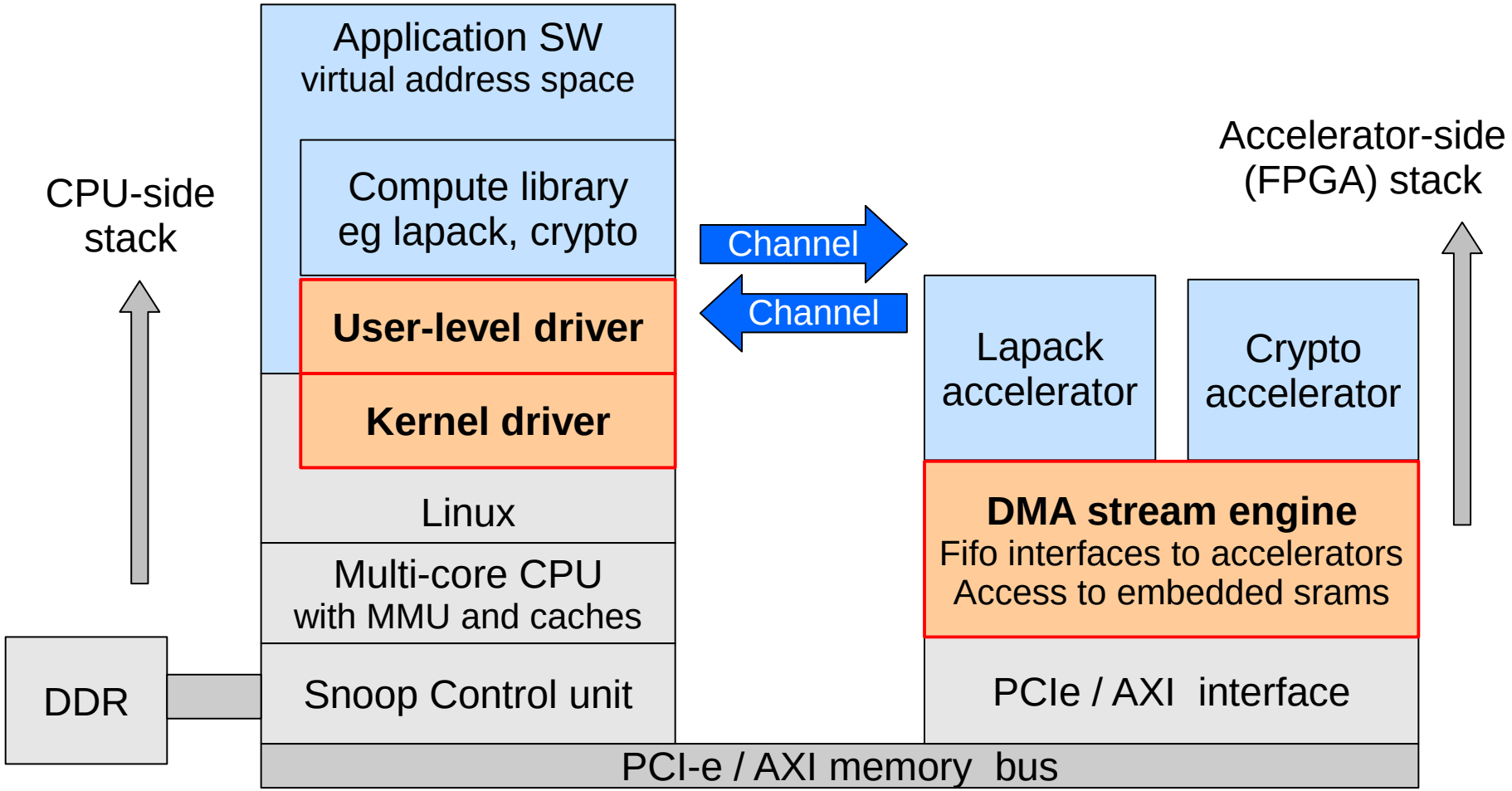
Buffered communication channel in software:

- Channel read/write blocks thread progress on empty/full fifo buffer.
- Implementation with 'semaphores' or 'monitors' causes the OS to sleep and wake-up the threads as needed.
- Channel data is fully cached for efficient CPU access, under HW cache-coherency control.

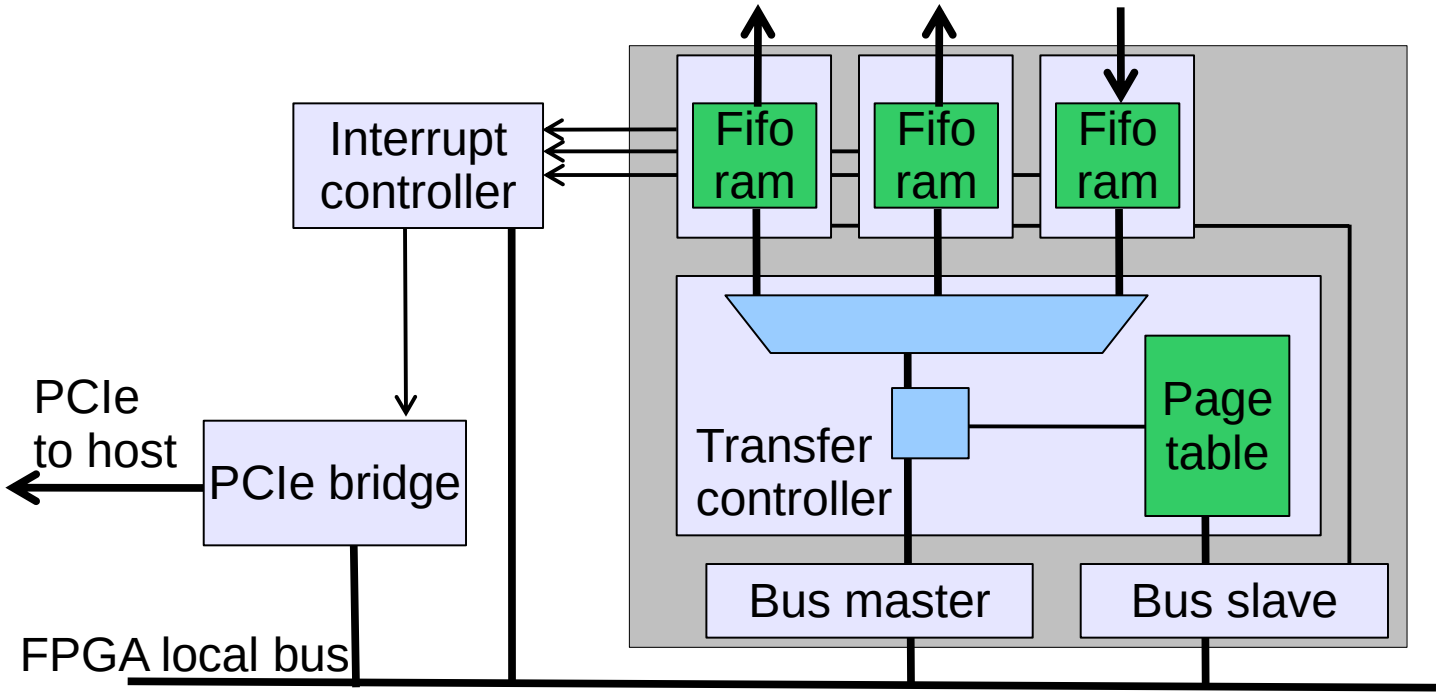
When one of the 'threads' is actually a hardware accelerator:

- Memory paging, coherency, and consistency issues
- Interact with the OS thread scheduler for wake-up

HW/SW communication stack



Streaming DMA engine in FPGA



- Fifo ram acts as cache for larger fifo buffer in host memory, performs **explicit cache control** (write-back, invalidate)
- Creates interrupts to wakeup selected software threads on host OS

Linux kernel driver

Traditional driver:

- Provides `open()`, `read()`, `write()` to application SW.
- Disadvantage: read/write are expensive.
  Require transition to 'kernel mode' on each call.

Up-to-date approach:

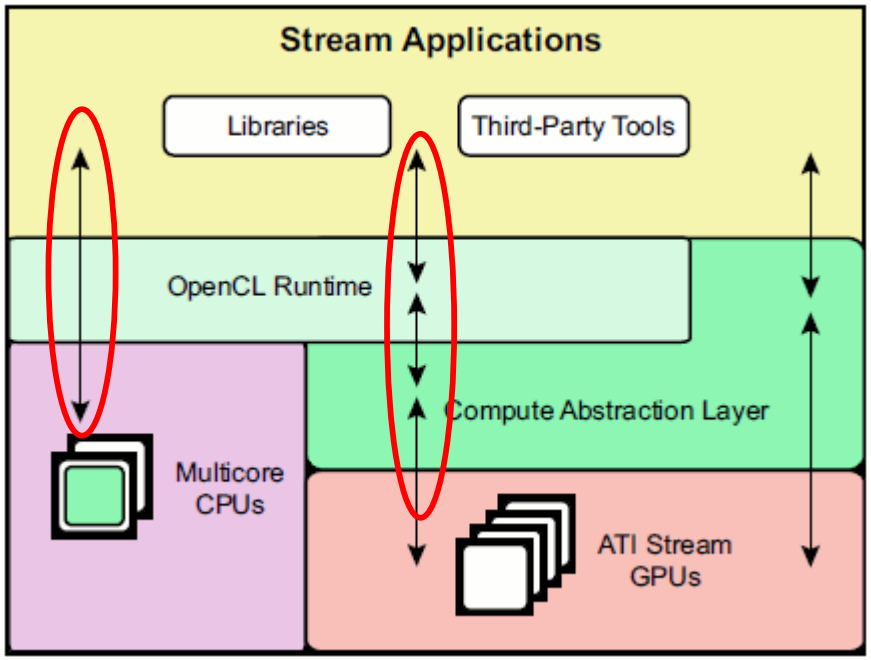
- Provide `mmap()`: application gets direct access to FPGA memory through pointer dereferencing. Typically **uncached**!

Furthermore:

- Provide Interrupt-Service-Routine for accelerator stream-engine.
  Signals Linux kernel to wake-up appropriate application thread.

Unfortunately, writing kernel-level device drivers is not popular among application SW developers nor among HW FPGA developers.

Streaming to GP-GPU accelerator



"clEnqueueMapBuffer()... *this is not an easy API call to use and comes with many constraints, such as page boundary and memory alignment*"

OpenCL 'Streaming':

- CPU and GPU process concurrently: buffered command queues
- Data exchange through PCIe shared memory space
- Relying on DMA support in the GPU device driver
- Application programming gets considerably complicated

Section 3 CPU <-> accelerator: final notes

CPU initiated communication

- CPU load/stores directly into device internal address space
- **Scalar load/stores** in uncached device memory space:
  High latency, low bandwidth
- Device setup and control traffic!

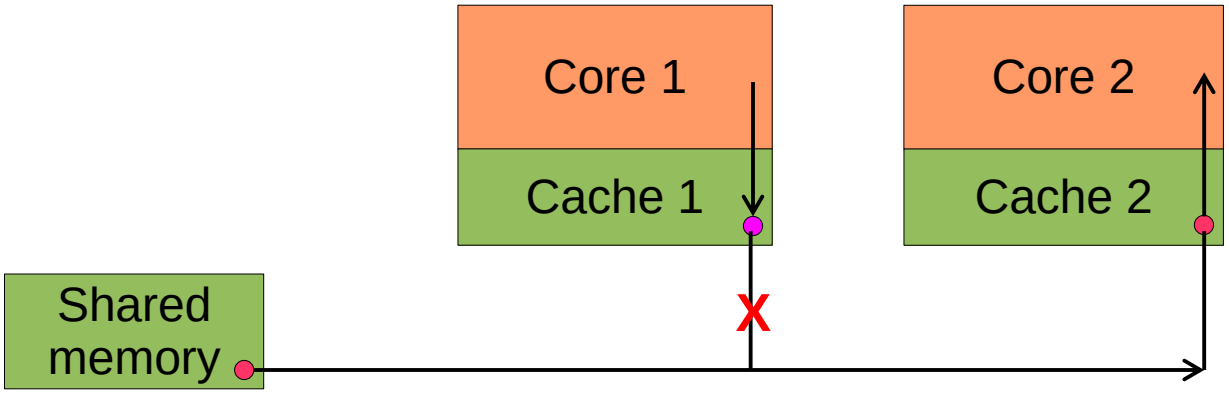
Accelerator initiated communication (DMA)

- **Burst-mode transfers** initiated by accelerator, into DDR space
- CPU load/store operations into local cache
- Cache-coherent through CPU snooping support
- High bandwidth, streaming data, efficient bus utilization!

Section 4a: cache coherency

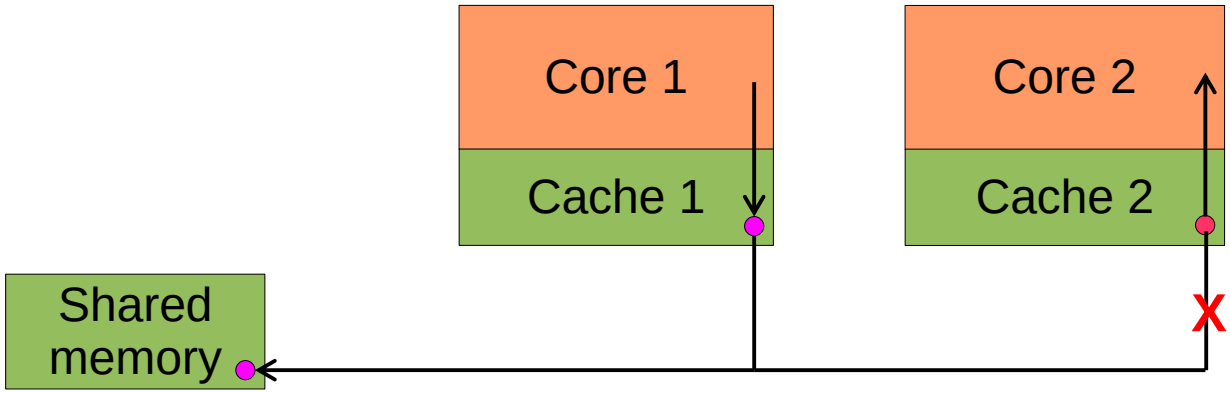
	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	Section 2
System implementation view	Section 3	**Section 4a**

No cache coherency? problem example 1



- Core 1 writes value 'a' to address P. It has a 'write-back' cache policy, so the value will stick in its cache, is not yet flushed to memory.
- Core 2 reads from address P. This address is not yet in its cache, the cache miss fetches the value from memory. The read delivers an outdated (wrong) value, since 'a' was not in memory.

No cache coherency? problem example 2



- Core 1 writes value 'a' to address P. The value (the cache line) gets somehow flushed to memory.
- Core 2 reads from address P. This address appears in its cache. The data in the cache-line is outdated. The read delivers a wrong value.

Cache coherency issues do NOT occur inside a dual-core ARM, or inside a multi-core / multi-cpu Intel machine, thanks to *HW cache coherency*.

Issues DO occur between the ARM and the DSP inside OMAP-DaVinci

Cache coherency - industry support (1)

- Defacto standard for **homogeneous** multi-core processors
- The **processor centric** view through decades of computer architecture history, resulted in marginal support for cache-coherent co-processors/accelerators:
  - PCI(-e) protocols do **not** have (symmetrical) cache-coherency support.
  - The cache coherency features of IBMs 'CoreConnect' did **not** make it into Xilinx' subset of its PLB bus.
  - The cache coherency extensions of ARM's AMBA 4 bus were postponed for a long time, and did **not** yet make it into Xilinx' "Zync" series or TI's "OMAP" series!

Cache coherency - industry support (2)

Current industry standard: **one-way** coherency

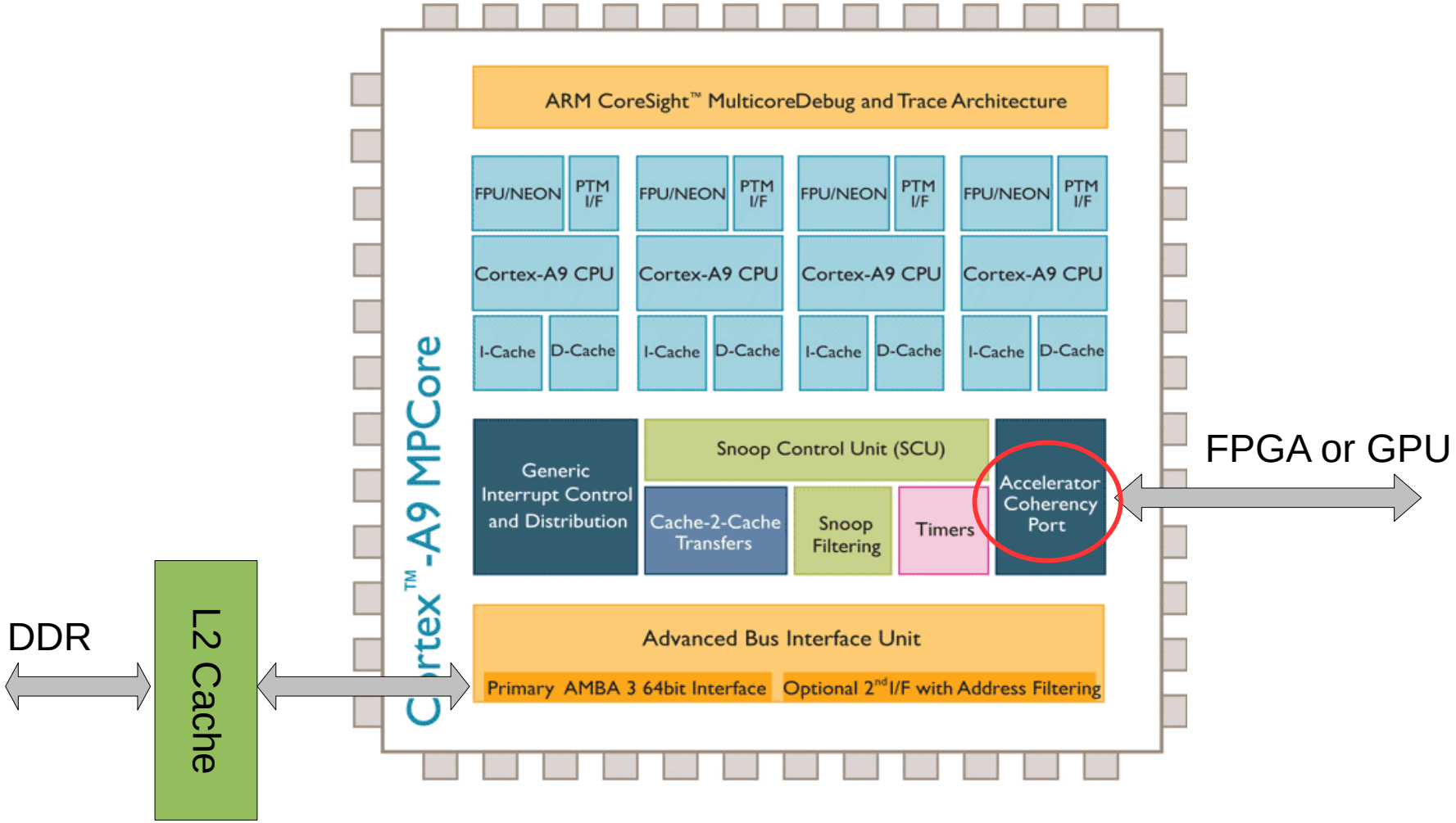
- Multi-core CPUs are mutually cache-coherent
- Co-processor traffic to shared memory 'snoops' processor caches, for both reads and writes.
- Co-processor caches are presumed absent: CPU-to-memory traffic ignores Co-processors

Examples:

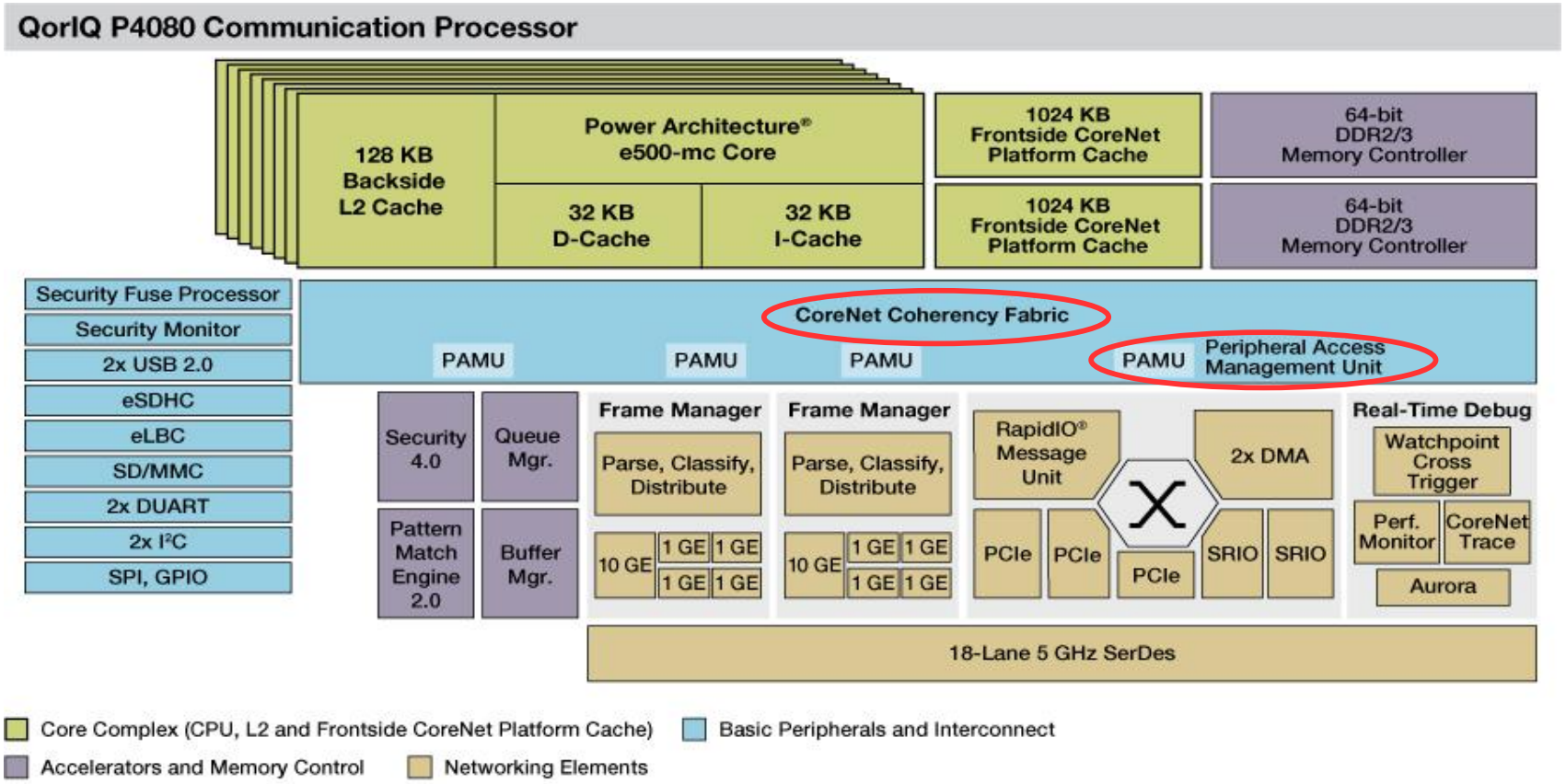
- PCI(-e) traffic passes through CPU memory controller.
- ARM cortex MPCore in Xilinx' Zync with its AXI *'Accelerator Coherency Port'* towards the FPGA fabric

Next generation products might adopt ARM's AMBA 4 ACE "*CoreLink Cache Coherent Interconnect*" for symmetrical coherency (first instance: ARM's "*Big-Little*" strategy)

ARM (A9) multicore example

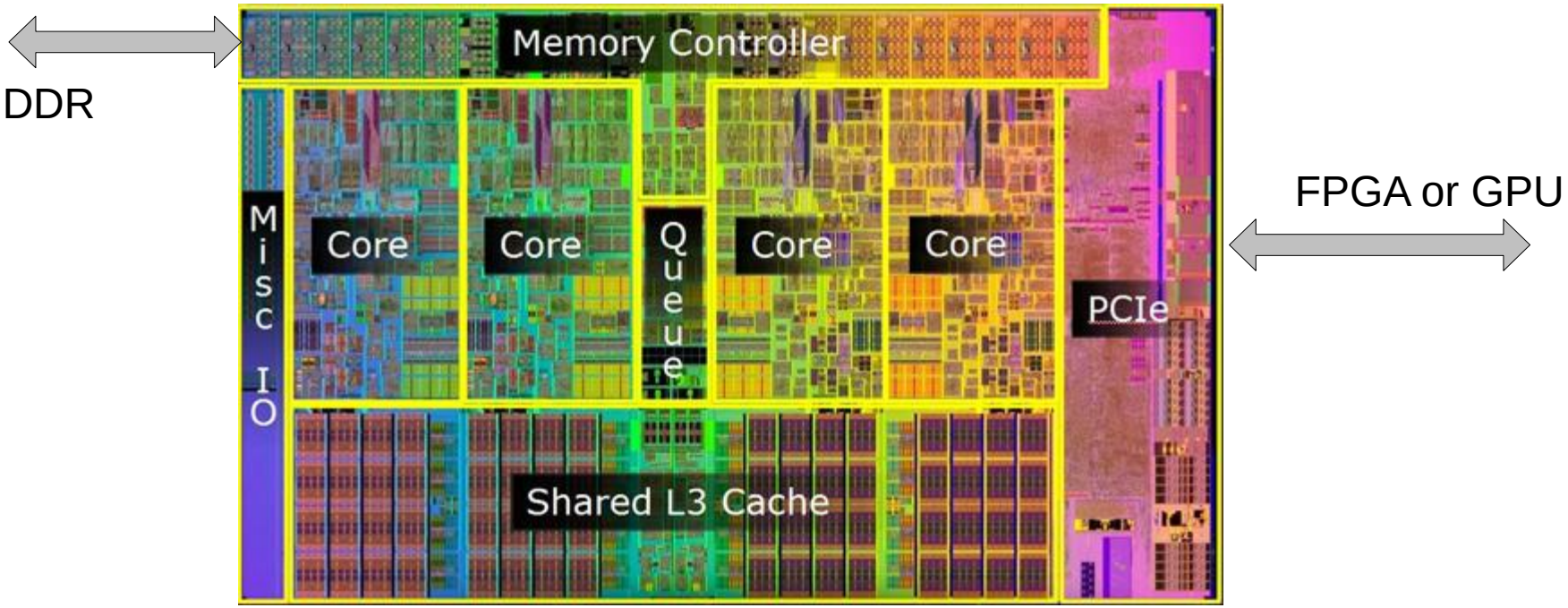


Freescale multicore example



Cache-coherent fabric, peripherals have a 'reduced' interface

Intel (i5) multicore example



- Device reads will be pulled from CPU L1/L2/L3 caches ☺
- Device writes first flush & invalidate matching CPU cache lines to DDR
  then finish device writes to DDR ☹
- PCIe 3.0 improves on writes with new caching hints in the protocol

Cache coherency - statements

- Hardware cache coherency is required for application multi-thread libraries and multi-core OS support.
- HW-CC is decades old and proven technology. For CMP (chip multi-processors) there is no sufficient reason to omit this.
- HW-CC is not yet found in heterogeneous (embedded) systems. Due to HW architects underestimating SW consequences…

Section 4a: memory consistency

	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	Section 2
System implementation view	Section 3	**Section 4b**

Memory consistency: read/write ordering

Different models/contracts that specify memory orderings. E.g.:

- Sequential consistency:
  - All processors observer memory updates *to a particular cell or page* as occuring in the same order.
  - Writes from a single processor are observed in issue order.
  - No guarantee on the interleaving from different processors, nor about different memory locations.
- Pipelined consistency (weaker then 'sequential'):
  - Writes from a single processor are observed in issue order.
  - Interleavings from different processors might be seen differently by other processors.

A solid platform specification is required for programmability!

Memory consistency / ordering

Processor A:

```
A = compute_result();
Flag = 1;
```

Processor B:

```
while (!Flag); // wait for data
use_result(A);
```

- Such SW can easily produce wrong results due to reordering:
  - When compiler re-orders instructions
  - When CPU does out-of-order instruction execution
  - When `Flag` is in a faster section of memory then `A`.

- Prevent instruction re-ordering, and/or insert **'memory barriers'**:

```
A = compute_result();
__sync_synchronize();
Flag = 1;
```

```
while (!Flag); // wait for data
__sync_synchronize();
use_result(A);
```

- Was weakly specified in C, subtle differences among compilers. C++11 has standardized support.
- Easily introduces bugs that are very hard to find!

Todays state...

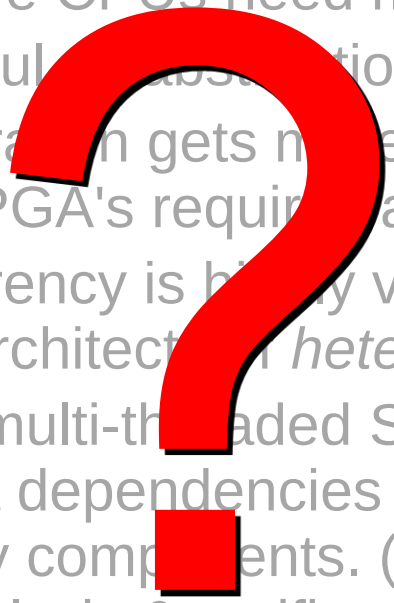
- ARM: “Memory coherency in a Cortex-A9 MPCore is maintained following a weakly ordered memory consistency model.”
- This is similar to PowerPC and Mips architecture.
- Intel is growing its number of cores with strong consistency.

Conclusion

- Today’s multi-core CPUs need multi-threaded applications.
- SW threads useful as abstraction for accelerator functionality.
- GP-GPU acceleration gets more popular then FPGA accelerators.
  Application of FPGA's requires a broad range of capabilities.
- HW cache-coherency is highly valuable for the SW programmer, but still ignored by architects of *heterogeneous* SoCs.
- Hand-writing of multi-threaded SW is highly error-prone.
  Inter-thread data dependencies are easily overlooked,
  e.g. inside library components. (C++ STL containers!)
  Use tools for analysis & verification!

Questions?

- Today's multi-core CPUs need multi-threaded applications.
- SW threads useful abstraction for accelerator functionality.
- GP-GPU acceleration gets more popular then FPGA accelerators. Application of FPGA's require a broad range of capabilities.
- HW cache-coherency is highly valuable for the SW programmer, but still ignored by architects of *heterogeneous* SoCs.
- Hand-writing of multi-threaded SW is highly error-prone. Inter-thread data dependencies are easily overlooked, e.g. inside library components. (C++ STL containers!) Use tools for analysis & verification!



Questions?

	Functional pipeline partitioning	Data parallel partitioning
Software Application view	Section 1	ion 2
System implementation view	Section 3	Section 4

Thank you

Check www.vectorfabrics.com for a free demo on concurrency analysis