Master Program (Laurea Magistrale) in Computer Science and Networking

High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

4. Shared Memory Parallel Architectures

4.5. Multithreading, Multiprocessors, and GPUs

Contents

- Main features of explicit multithreading architectures
- Relationships with ILP
- Relationships with multiprocessors and multicores
- Relationships with network processors
- GPUs

Basic principle

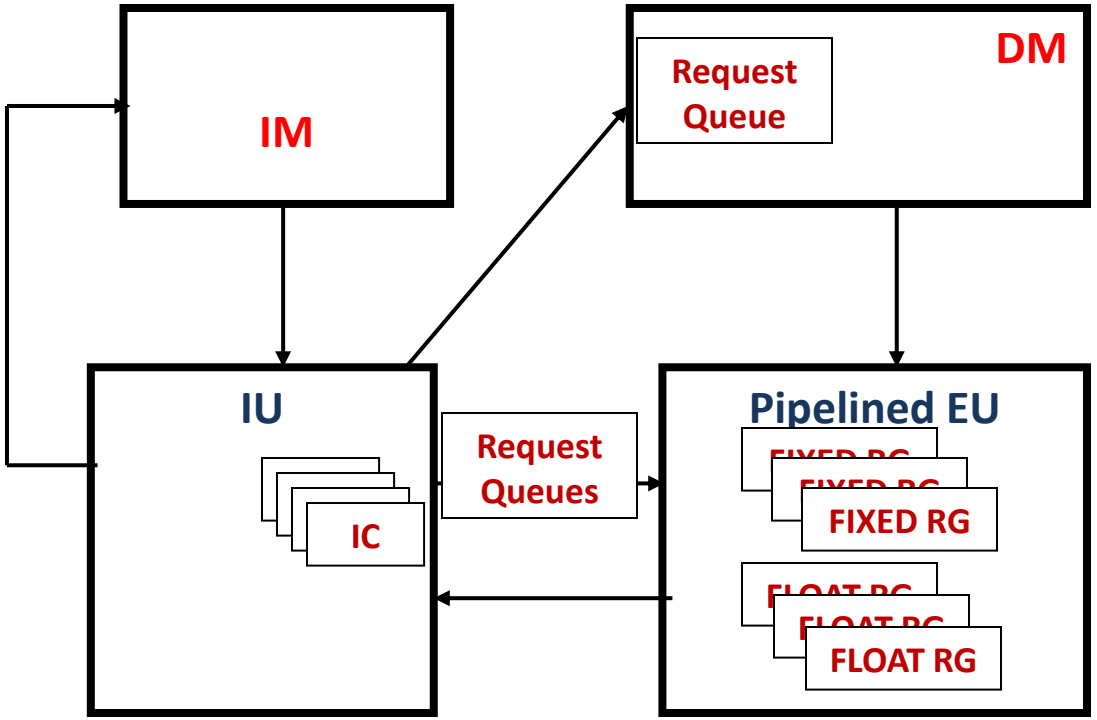
- Concurrently execute *instructions of different threads of control* within a *single pipelined processor*
- Notion of ***thread*** in this context:
  - NOT "software thread" as in a multithreaded OS,
  - *Hardware-firmware supported thread:* an independent execution sequence of a (general-purpose or specialized) processor:
    - A process
    - A compiler generated thread
    - A microinstruction execution sequence
    - A task scheduled to a processor core in a GPU architecture
    - Even an OS thread (e.g. POSIX)
- In a multithreaded architecture, a thread is the *unit of instruction scheduling for ILP*
  - *Neverthless, multithreading can be a powerful mechanism for* ***multiprocessing*** *too (i.e., parallelism at process level in multiprocessors architectures)*
- (Unfortunately, there is a lot of confusion around the word "thread", which is used in several contexts often with very different meanings)

Basic architecture

- *More independent program counters*
- *Tagging mechanism* (unique identifiers) to distinguish instructions of different threads within the pipeline
- Efficient mechanisms for *thread switching:* very efficient context switching, from zero to very few clock cycles
- *Multiple register sets*
  - (not always statically allocated)

Interleave the execution of instructions of different threads in the same pipeline.

Try “to fill” the latencies as much as possible.



Basic goal: latency hiding

- ILP: latencies are sources of performance degradations because of data dependencies
  - Logical dependencies induced by “long” arithmetic instructions
  - Memory accesses caused by cache faults
  - Idea: *interleave instructions of different threads to increase distance*

- Multithreading and data-flow: similar principles
  - when implemented in a Von Neumann machine, multithreading requires multiple contexts (program counter, registers, tags),
  - while in a data-flow machine every instruction contains its “context” (i.e., data values),
  - the data-flow idea leads to multithtreading when the assembler level is imperative.

Basic goal: latency hiding

- Multiprocessors: remote memory access latency (interconnection network, conflicts) and software lockout
  - Idea: *instead of waiting idle for the remote access conclusion, the processor switches to another thread* in order to fill the idle times
  - *Context switching (for threads) is caused by remote memory accesses too*
  - Exploit multiprogramming of a single processor with a much finer grain
- *Context switching for threads* is very fast: **multiple contexts** (program counter, general registers) are present inside the processor itself (not to be loaded from higher memory levels), and no other "administrative" information has to be saved/restored
  - Multithreading is NOT under the OS control, instead it is implemented at the **firmware level**
- This is compatible with multiprogramming / multiprocessing: **process states** still exist (*context switching for processes is distinct from context swiching for threads*)

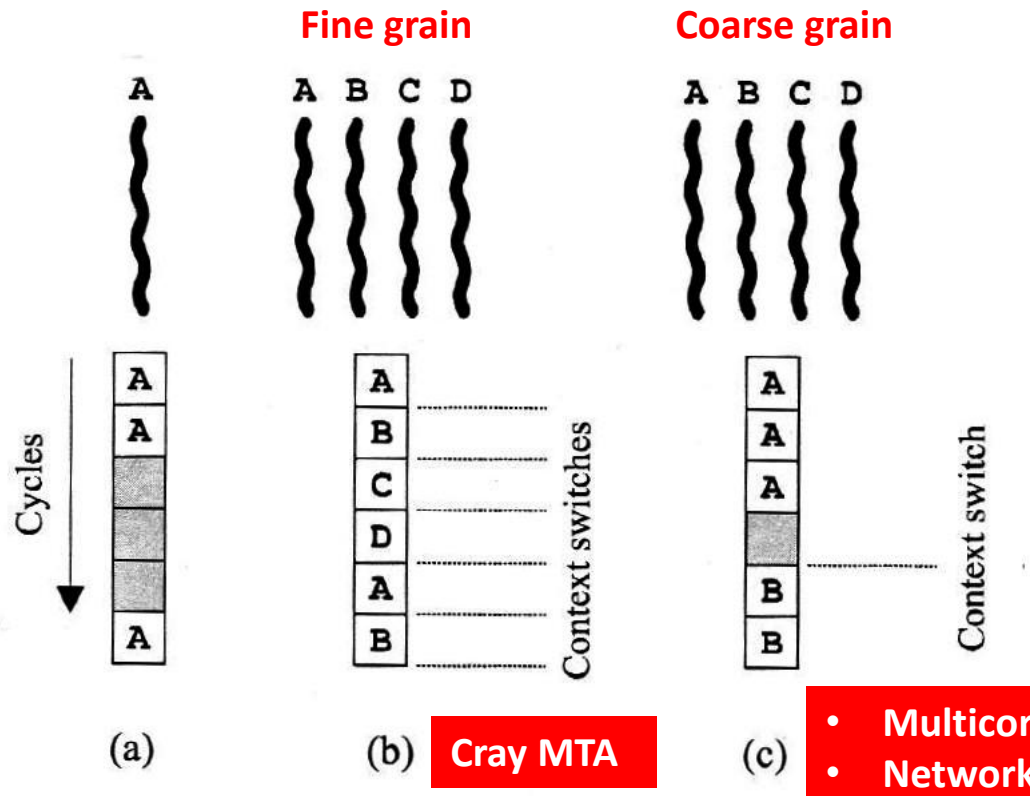
Taxonomy of multithreaded architectures

[Ungerer, Robic, Silc]: course references

Instructions in a given clock cycle can be issued from

- a single thread
  - **Interleaved multithreading** (**IMT**)
    - an instruction of *another* thread is fetched and fed into the execution pipeline (of a *scalar* processor) at each clock cycle
  - **Blocked multithreading** (**BMT**)
    - the instructions of a thread are executed successively (in pipeline on a *scalar* processor) *until an event occurs that may cause latency* (e.g. remote memory access); this event induces a context switch
- multiple threads
  - **Simultaneous multithreading** (**SMT**)
    - instructions are simultaneously issued from multiple threads to the execution units of a ***superscalar*** processor, i.e. superscalar instruction issue is combined with the multiple-context approach

Single-issue processors (scalar: pipelined CPU)



Cray MTA

- Multicore with many "simple" CPUs
- Network processors

Fig. 2. Different approaches possible with single-issue (scalar) processors: (a) single-threaded scalar, (b) interleaved multithreading scalar, (c) blocked multithreading scalar.

Multiple-issue processors (superscalar CPU)

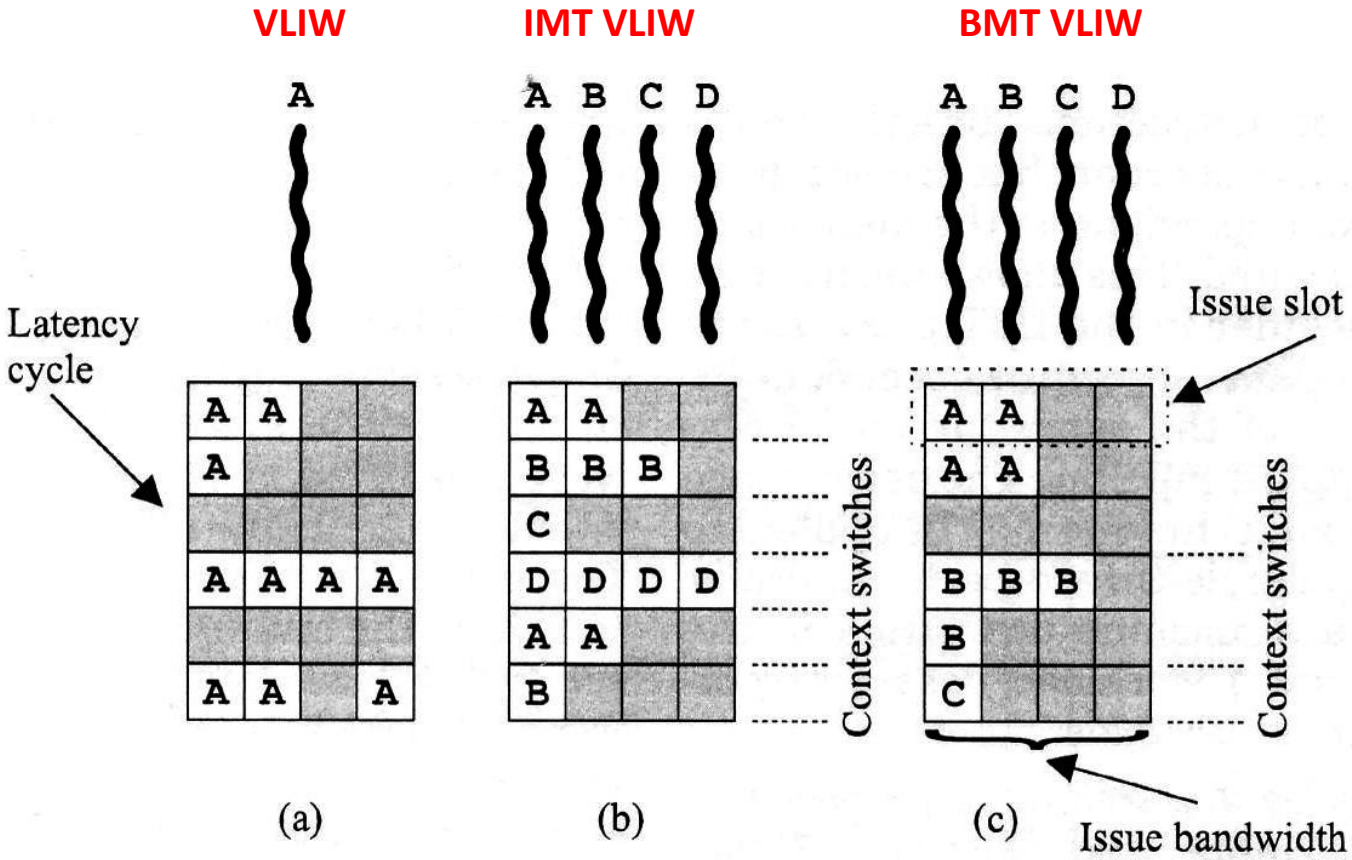


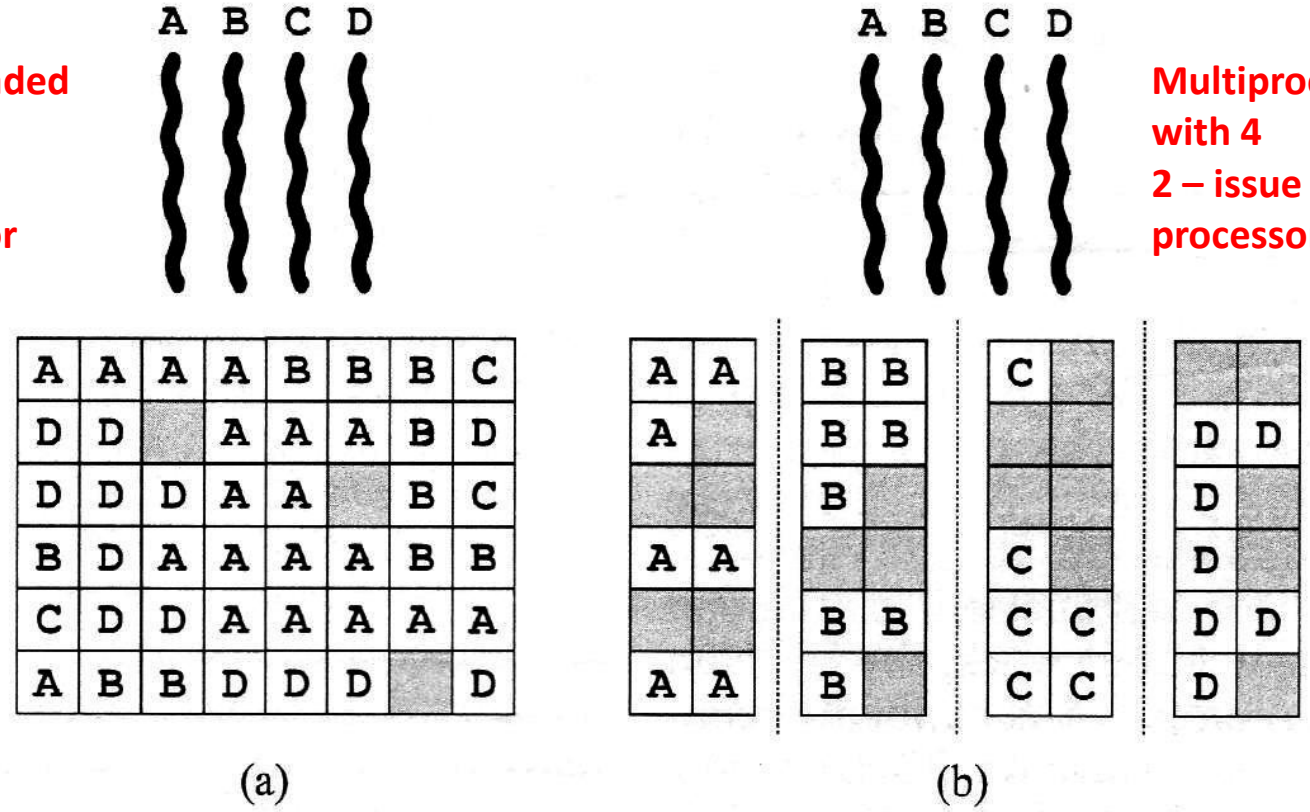
Fig. 3. Different approaches possible with multiple-issue processors: (a) single-threaded superscalar, (b) interleaved multithreading superscalar, (c) blocked multithreading superscalar.

b) and/or c):

- **Blue Gene**
- **SUN UltraSPARC**
- **Intel Xeon Hyperthreading**
- **GPU**

Simultaneous multithreading *vs* multicore

4 – threaded
8 – issue
SMT
processor



(a) (b)

Multiprocessor
with 4
2 – issue
processors

Fig. 4. Issuing from multiple threads in a cycle: (a) simultaneous multithreading, (b) chip multiprocessor.

Latency hiding in multiprocessors

- Try to exploit memory hierarchies at best
  - Local memories, NUMA
  - Cache coherence
- Communication processor
  - Interprocess communication latency hiding
- Additional solution: multithreading
  - Remote memory access latency hiding
- Fully compatible solutions

  e.g. KP is multithreaded in order to "fill" KP latencies for remote memory accesses,

  thus, KP is able to execute more communications concurrently: for each new communication request, a new KP thread is executed, and more threads share the KP pipeline,

  thus, **increased KP bandwidth.**

Which performance improvements?

- Improving CPU (CPUs) *efficiency* (i.e. utilization, ε)
- What about *service/completion time* for a single process ? Apparently, no direct advantage,
- *but in fact:*
  - communication / calculation overlapping through KP
  - **increased KP bandwidth**: this leads to an improvement in service time and completion time of parallel programs,
- *and:*
  - improvement of **ILP** performance, thus some improvemet of program completion time.
  - *Best situation*: *threads belong to the same process*, i.e. a process is further parallelized through threads;
  - meaningful improvement in service/completion time, provided that: **high parallelism is exploited between threads of the same process.**
  - Here we can see the **convergence between multithreading and data-flow**: exploit data-flow parallelism between threads of the same process (**data-flow multithreading**).
  - **Research issue: multithreading optimizing compilers.**

Excess parallelism

- The idea of multithreading (BMT) for multiprocessors can have another interpretation:
  - Instead of using
    1. ***N*** **single-threaded processors**

    use
    2. ***N/p*** **processors, each of which is** ***p-threaded***
- (Nothing new *wrt* the old idea of multiprogramming: except that context switching is no more a meaningful overhead)
- *Under which conditions performances (e.g. completion times) of solutions 1 and 2 are comparable?*

  Despite the increasing diffusion of multithreaded architectures: still an open research problem.

Excess parallelism

Rationale:

- A data-parallel program is designed for *N virtual processors*,
  - where the virtual processors are chosen with the goal of achieving the maximum parallelism for a "perfect" architecture (e.g., zero communication latency).
- Its implementation exploits *N/p* real processors,
  - where *p* is *partition size* of the real processors solution *wrt* the one of the virtual processors solution.
- In several cases (not always), the *order of magnitude* of completion time is not increased:
  - this guarantees that the program "scales" well.
- *Conceptually*, we can consider that the real processors solution exploits "*p* excess parallelism"
  - actually, the real solution exploits *N/ p* **sequential** workers;
  - **why not *N/p* parallel workers, each worker with *p* excess parallelism? i.e. *p* parallel threads per worker.**
  - Example: a *map*.

Excess parallelism

From the complexity theory of parallel computations (PRAM), we know that under general *conditions excess parallelism doesn't increase the order of magnitude of completion time*:

- Context: data-parallel programs executed on a shared memory architecture with logarithmic network, i.e.

$$base\ latency = O(\log N)$$

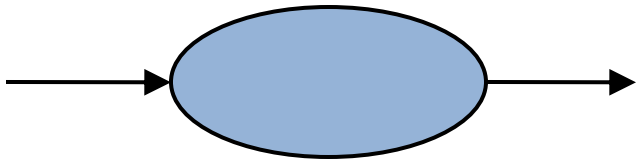
- "Optimal" parallel algorithms exploits the architecture in such a way that:

$$under\text{-}load\ latency = O(\log N)$$

- This can be achieved also with N/p processors, each of which with *excess parallelism* p, provided that p is chosen properly according to the algorithm and the architecture (not greater than $O(\log N)$).

Multithreading and communication: example

A process working on stream:



```
while true do
    { receive (…, a);
      b = F (a);
      send ( …, b)
    }
```

Let's assume *zero-copy* communication: from the performance evaluation viewpoint, process alternates calculation (latency T_{calc}) and communication (latency T_{send}).

Without no communication processor nor multithreading:

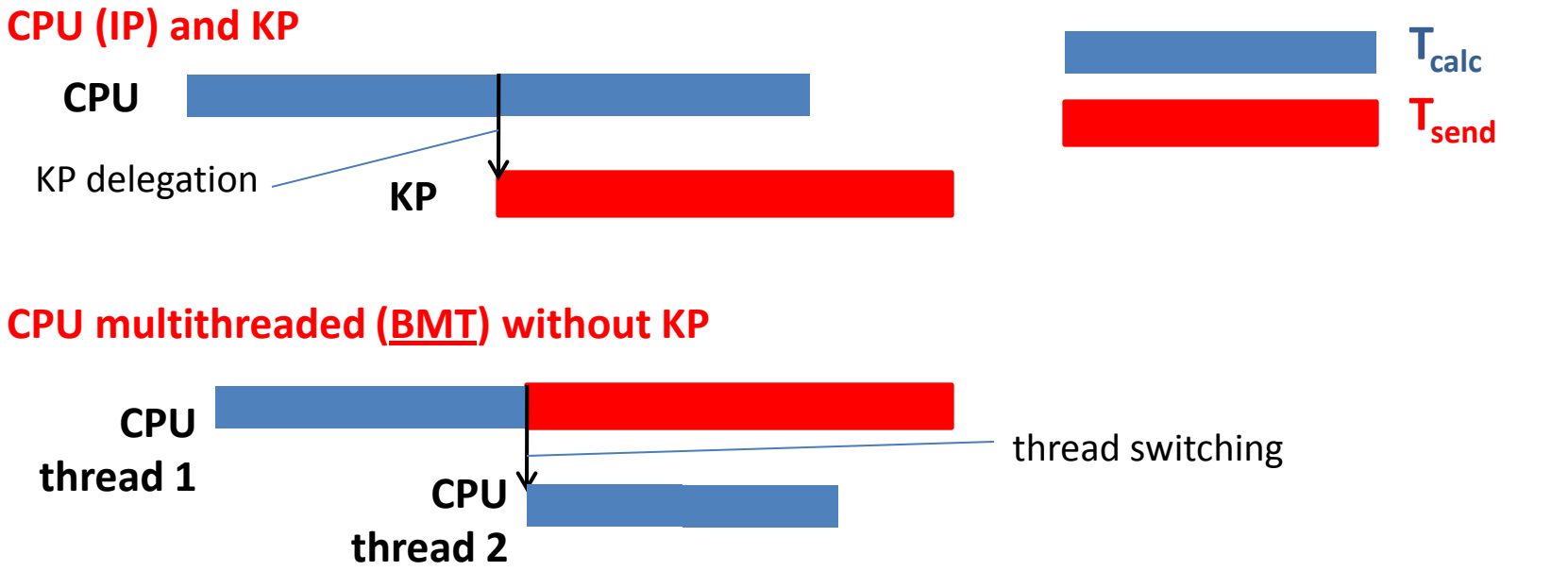
$$T_{service} = T_{calc} + T_{send}$$

In order to achieve (i.e. *masking communication latency*):

$$T_{service} = T_{calc}$$

we can exploit parallelism between calculation and communication in a proper way. This can be done by using *communication processor and/or multithtreading* according to various solutions.

Example: behiavioural schematizations



In distributed memory machines, a *send* invocation can cause a *thread switching*.

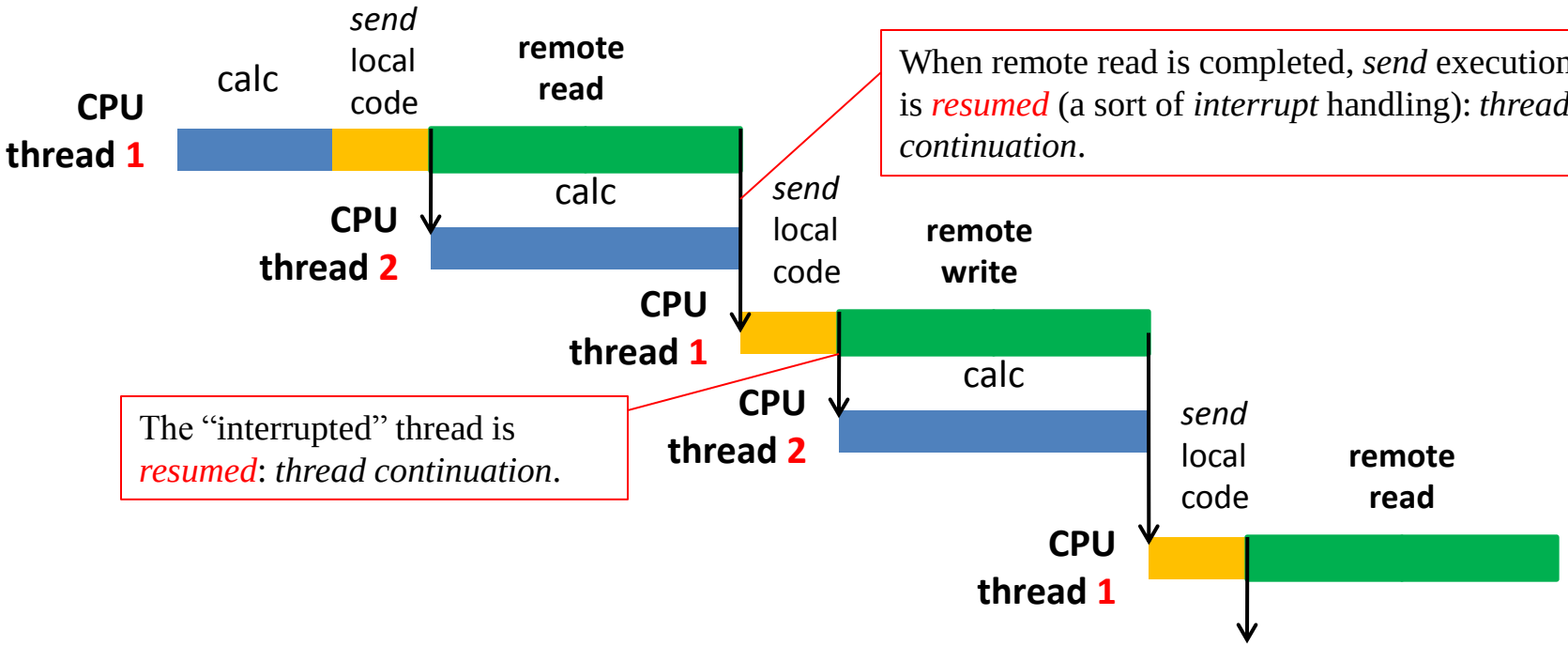
If a shared memory machine, the **real behaviour is**: during a *send* execution, thread switching occurs several times, i.e. each time a remote memory requests is done.

Because the majority of *send* delay is spent in remote memory accesses, **equivalently (from the performance evaluation modeling)** we can assume that thread switching occurs, just one time, when a *send* is invoked (**approximate** equivalent behaviour).

In other words, this is the behaviour of the **abstract architecture**.

Approximate equivalent behaviour in multithreaded CPU

Real behaviour: interleaving calculation and *send* execution



Equivalent behaviour (cost model) : best case approximation

Observations

- Simplified cost model:
  - taking into account of the very high degree of nondeterminism and interleaving, that characterizes multithreaded architectures
  - for distributed memory architectures, this cost model has a better approximation

- Implementation of a thread-suspend / -resume mechanism at the firmware level
  - *in addition to* the all the other pipeling/superscalar synchronizations
  - additional complexity of the hardware-firmware structure

Example: $T_{calc} \geq T_{send}$

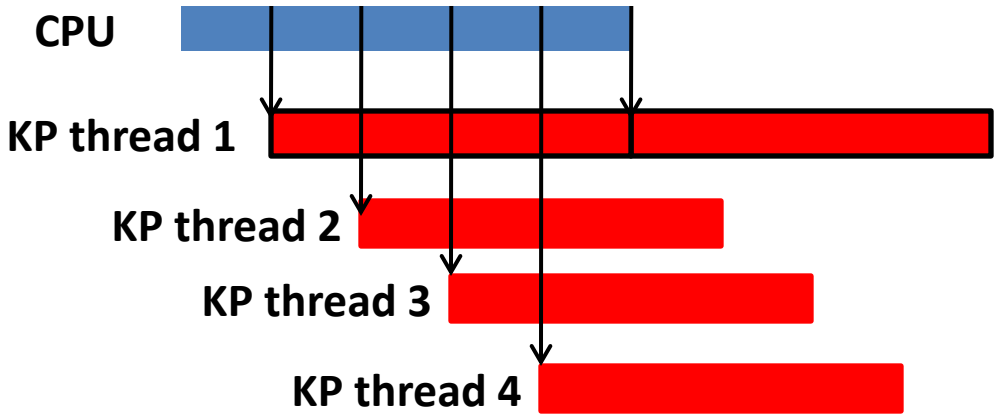
CPU (IP) scalar, KP scalar (1-issue multithreaded)

CPU 2-issue multithreaded, no KP

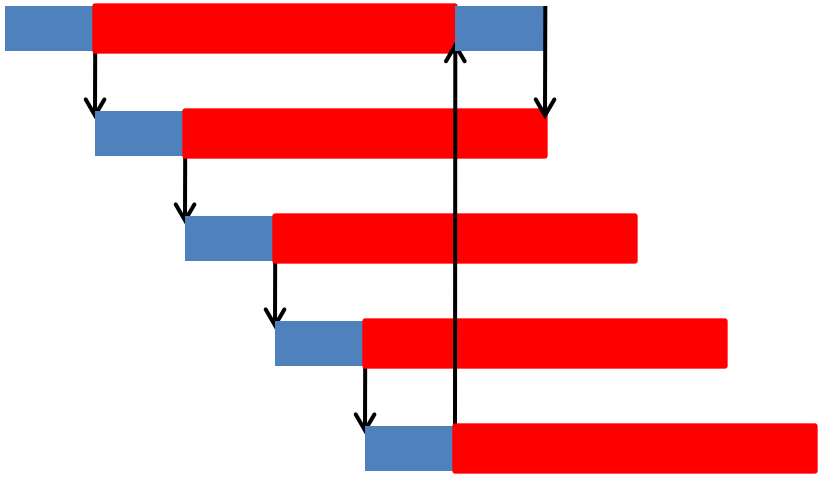
Equivalent service time ($\mathbf{T_{calc}}$).
Equivalent parallelism degree per node: *two real-parallelism threads, on the same node, correspond to two non-multithreaded nodes*. In fact, many hardware resources are duplicated in a 2-issue multithreaded node.
In principle, chip area is equivalent (hardware-complexity of the same order of magnitude). However, in practice … (see slide + 2).

Example: $T_{calc} < T_{send}$

CPU (IP) scalar, KP 4-issue multithreaded



CPU 5-issue multithreaded, no KP



Equivalent service time (T_{calc}).

Equivalent parallelism degree per node:

$$1 + \left\lceil \frac{T_{send}}{T_{calc}} \right\rceil$$

provided that

memory bandwidth > $\left\lceil \frac{T_{send}}{T_{calc}} \right\rceil$

(block) accesses per memory clock cycle.

This is the *upper bound to the parallelism per node* that can be exploited.

Observation: KP or not KP ?

The sevice time and the total parallelism degree per node being equal,

1) solution with CPU (IP) + *p*-threaded KP

has the same hardware-complexity of

2) solution with (*1 + p*)-threaded CPU, without KP.

However, in terms of real cost, solution 1) is *cheaper*,

e.g. it has a simpler hardware-firmware structure (less inter-thread synchronization in CPU pipelining, lower suspend/resume nesting), thus it has a lower power dissipation.

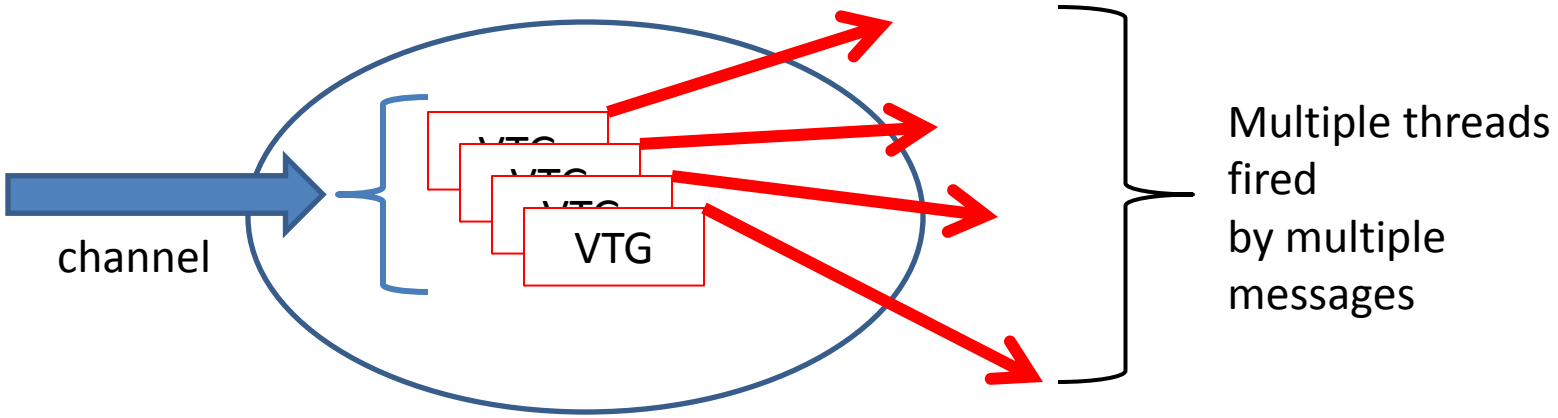
Moreover, solution 1 can be seen as just another rationale for *heterogeneous* multicore (main CPU + *p* cores).

Observation: parallel communications and parallel program optimization

- *Multithreaded CPU*, or *CPU + multithreaded KP*, is a solution to eliminate / reduce potential bottlenecks in parallel programs, provided that the memory bandwidth is adequate.
- Example: a *farm* program where *interarrival time* $T_A < T_{send}$
  - Emitter could be a bottleneck ($T_{emitter} = T_{send}$)
- Example: a *data-parallel* program when the Scatter functionality could be a bottleneck
  - Scatter service time > T_A
- In both cases, bottlenecks prevent to exploit the ideal parallelism solution (T_{calc} / T_A workers).
- In both cases, parallelization of communications eliminates / reduces bottlenecks.

Observation: importance of advanced mechanisms for interprocess communication

- Multithreading = parallelism exploitation and management (context switching) at firmware level
  - Efficient mechanisms for interprocess communication are needed
  - User level
  - Zero-copy
- Example: multiple processing on target variables are allowed by the zero-copy communication



Network processors and multithreading

- Network processors apply multithreading to bridge latencies during (remote) memory accesses
  - Blocked multithreading (BMT)
  - Multithreading applied to cores that perform the data traffic handling
- Hard real-time events (i.e., deadline soluld never be missed)
  - Specific instruction scheduling during multithreaded execution
- Examples:
  - Intel IXP
  - IBM PowerNP

IBM Wire-Speed Processor (WSP)

- Heterogenous architecture
- 16 general-purpose multithreaded cores: PowerPC, 2.3 GHz
  - SMT, 4 simultaneous threads/core
  - 16 Kb L1 instruction cache, 16 Kb L1 data cache (8-way set associative), 64-byte cache blocks
  - MMU: 512-entry, variable page size
  - 4 L2 caches (2 MB), each L2 cache shared by 4 cores
- Domain-specific co-processors (accelerators)
  - targeted toward networking applications: packet processing, security, pattern matching, compression, XML
    - custom hardware-firmware components for optimizations
  - networking interconnect: four 10-Gb/s links
- Internal interconnection structure: partial crossbar
  - similar to a 4-ring structure, 16-byte links

WSP

IBM Journal of Res & Dev, Jan/Feb 2010, pp. 3:1 – 3:11.

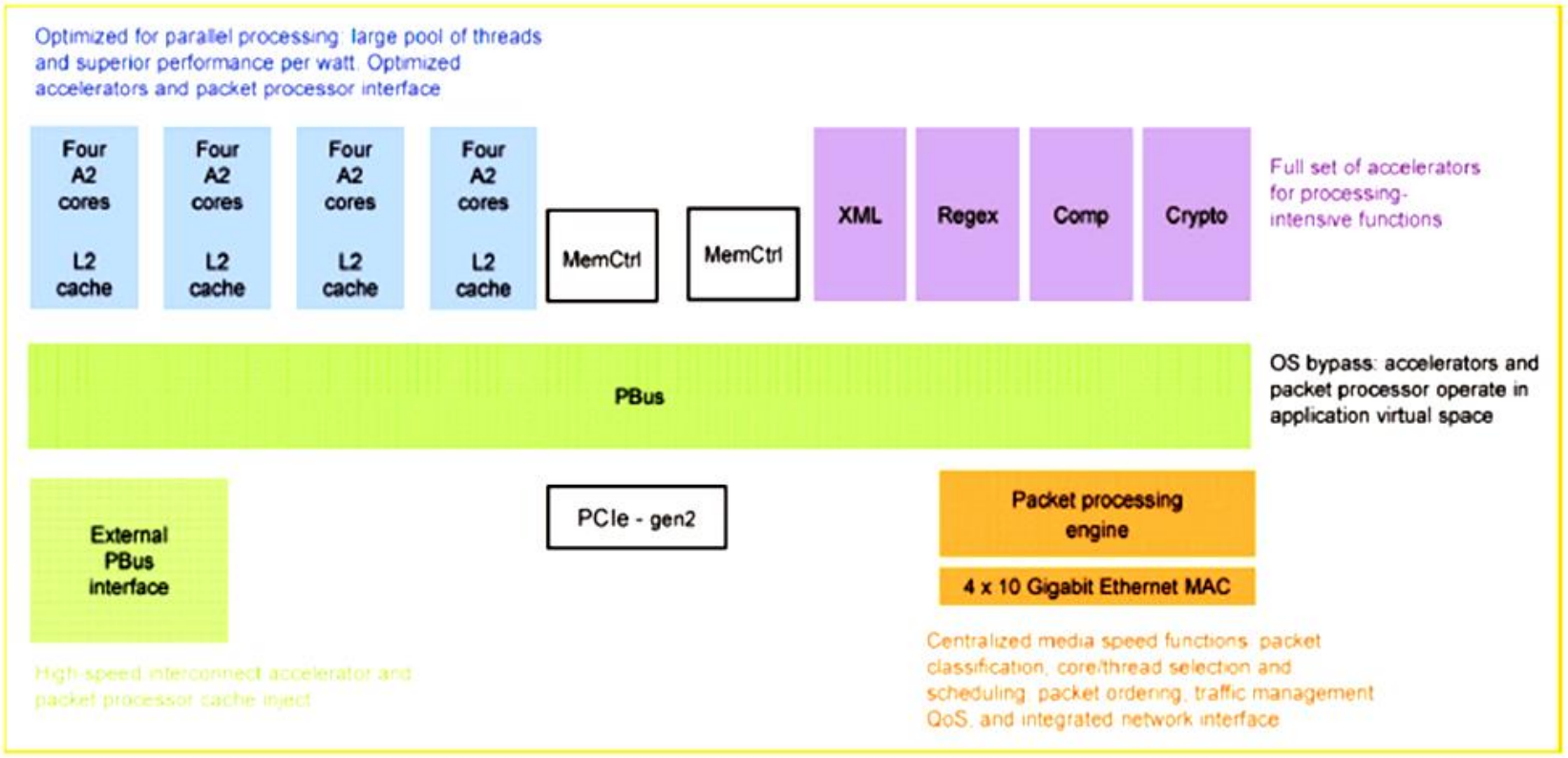


Figure 1

WSP functional diagram and characteristics. The A2 cores are described later in the text. (XML: Extensible Markup Language; Regex: regular expression and pattern-matching accelerator; Comp: compression and decompression accelerator; Crypto: cryptographic accelerator; MAC: media access control, a unique network adapter ID; MemCtrl: memory controller; QoS: quality of service; gen2: generation 2.)

WSP

Table 1 Functional specification of WSP accelerators. (AES: Advanced Encryption Standard; DES: Data Encryption Standard; TDES: Triple Data Encryption Standard; ARC4: Alleged Rivest Cipher 4; SHA: Secure Hash Algorithm; MD5: Message Digest 5; RSA: Public-Key Cryptography named after Ron Rivest, Adi Shamir, and Leonard Adleman; ECC: elliptic curve cryptography.)

Accelerator unit	Algorithm	No. of engines	Projected bandwidth	
			Typical	Peak
HEA	Network node mode	4	40 Gb/s	40 Gb/s
	Endpoint mode	4	40 Gb/s	40 Gb/s
Compression	Gzip (input bandwidth)	1	8 Gb/s	16 Gb/s
	Gunzip (output bandwidth)	1	8 Gb/s	16 Gb/s
Encryption	AES	3	41 Gb/s	60 Gb/s
	TDES	8	19 Gb/s	60 Gb/s
	ARC4	1	5.1 Gb/s	60 Gb/s
	Kasumi	1	5.9 Gb/s	60 Gb/s
	SHA	6	23–37 Gb/s	60 Gb/s
	MD5	6	31 Gb/s	60 Gb/s
	AES/SHA	3	19–31 Gb/s	60 Gb/s
	RSA and ECC (rates shown are for RSA 1,024-bit and RSA 2,048-bit keys)	3	45,000 and 7,260	60 Gb/s
XML	Customer workload	4	10 Gb/s	30 Gb/s
	Benchmark	4	20 Gb/s	30 Gb/s
RegX	For typical pattern sets	8	20–40 Gb/s	70 Gbps

WSP

Advanced features for *programmability + portability + performance*:

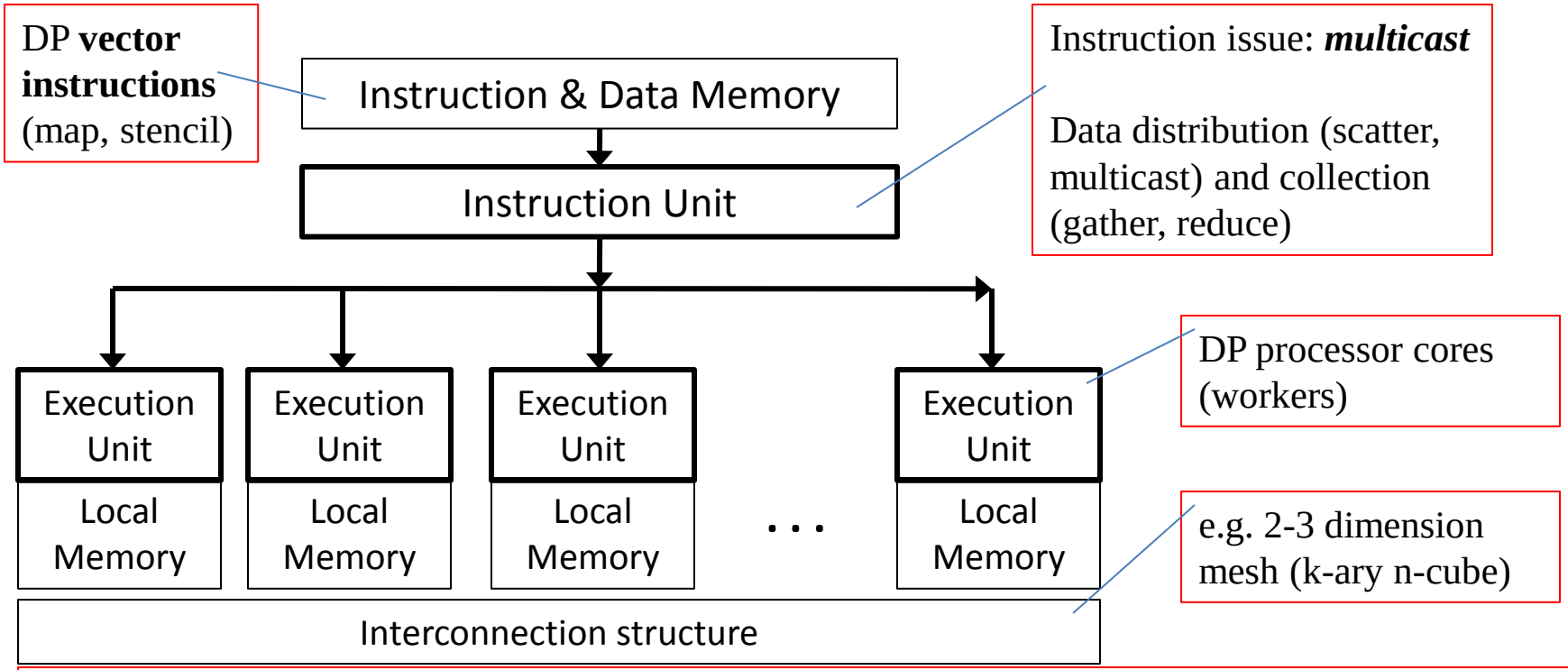
- Uniform addressability: uniform virtual address space
  - Every CPU core, accelerator and I/O unit has a separate MMU
  - Shared memory: NUMA architecture, including accelerators and I/O units (heterogeneous NUMA)
  - Coherent (snooping) and noncoherent caching support, also for accelerators and I/O
  - Result: accelerators and I/O are not special entities to be controlled through specialized mechanisms, instead they exploit the same mechanisms of CPU cores
    - *full process-virtualization of co-processors and I/O*
- Special instructions for locking and core-coprocessor synchronization
  - Load and Reserve, Store Conditional
  - Initiate Coprocessor
- Special instructions for thread synchronization
  - wait, resume

GPUs

- Currently, another application of the multithreading paradigm is present in GPUs (Graphics Processing Units) and their attempt to become “general” machines
- GPUs are SIMD machines
- In this context, threads are execution instances of data-parallel tasks (data-parallel workers)
- Both *SMT* and *multiprocessor + SMT* paradigms are applied

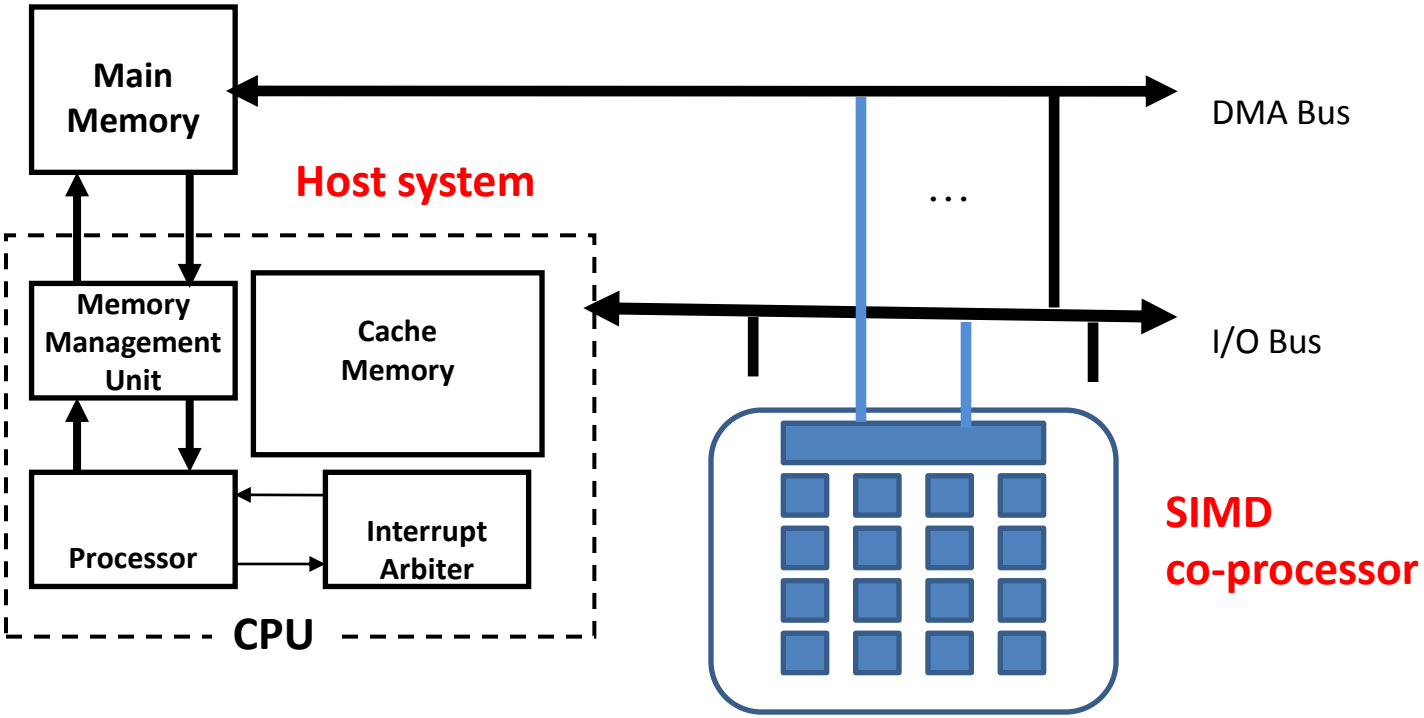
SIMD architecture

- SIMD (Single Instruction Stream Multiple Data Stream)
  - **Data-parallel** (DP) **paradigm at the firmware-assembler level**



- Example: IU controls the partitioning of a float vector into the local memories (scatter), and issues a request of "vector_float_addition" to all EUs
- Pipelined processing IU-{EU}, pipelined EUs
- Extension: partitioning of {EU} into disjoint subsets for DP multiprocessing (**MIMD + SIMD**)

SIMD: parallel, high-performance *co-processor*

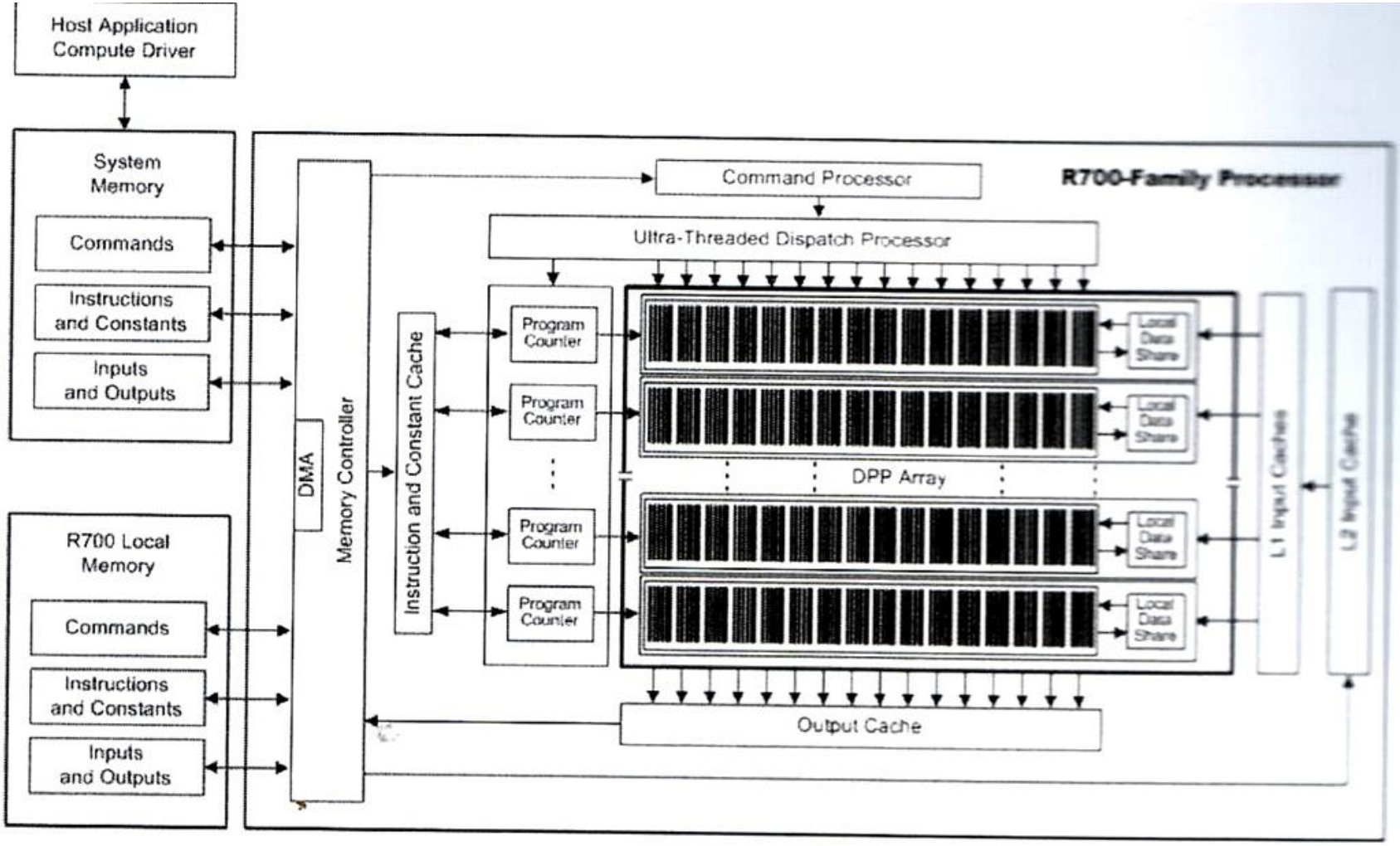


- SIMD cannot be general-purpose.
- *I/O bandwidth and latency for data transfer between Host and SIMD co-processor could be critical.*
- Challenge: proper utilization of central processors and peripheral SIMD co-processors for designing high-performance parallel programs

GPU parallel programs

- From *specialized* coprocessors for real-time, high-quality 3D graphics rendering (shader), to *programmable* data-parallel coprocessors
- *Generality vs performance? Programmability ?*
- *Stream-based SIMD Computing*: replication of stream tasks (shader code) and partitioning of data domain onto processor cores (EU)
- *Thread*: execution instance of a stream task scheduled to a processor core (EU) for execution
  - NOT to be confused with a software thread in multithreaded OS
  - *same meaning of thread in multithreaded architectures*.

Example of GPU: AMD



AMD GPU

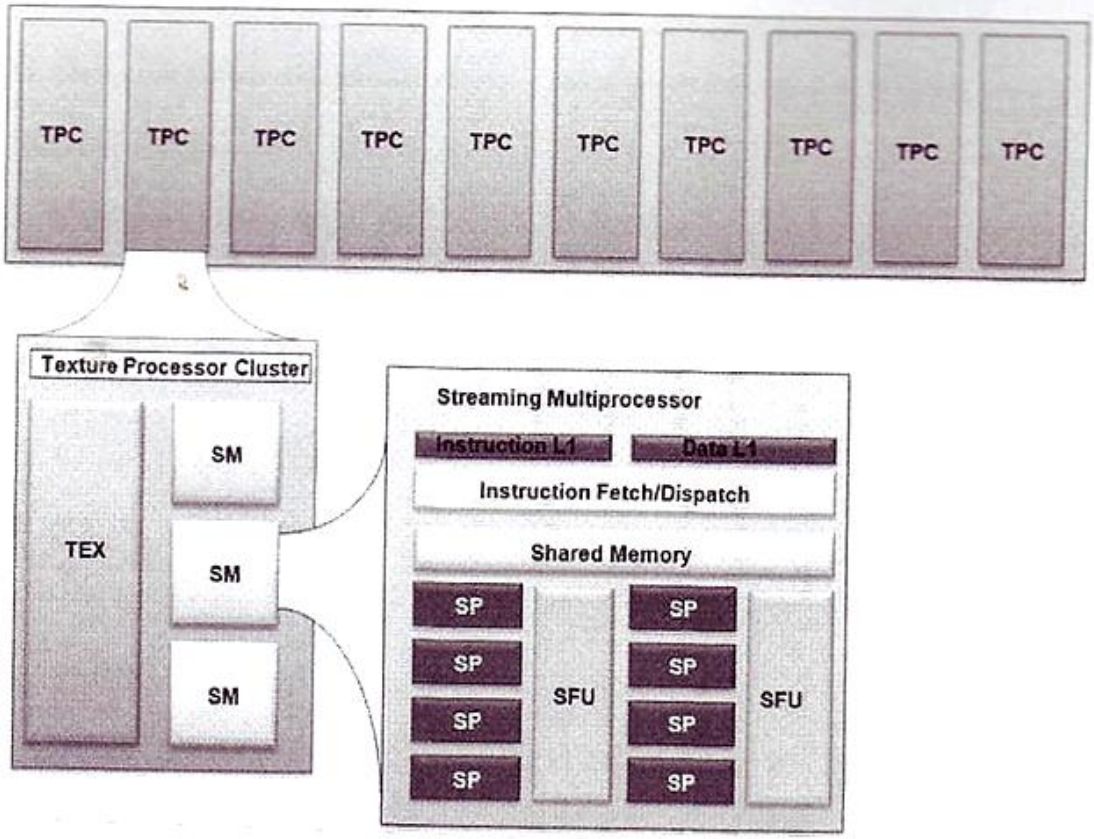
RV770

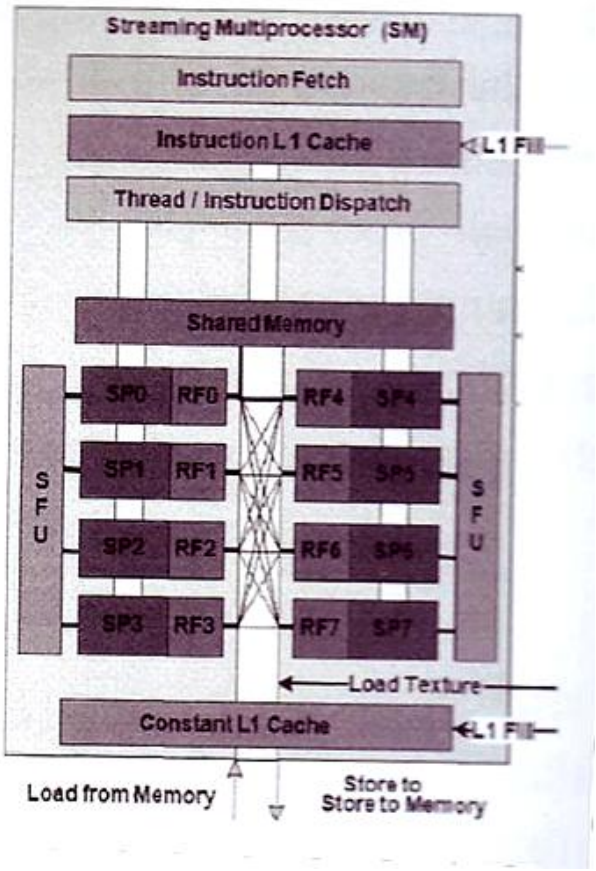
- {EU} is organized into 10 partitions
- Each EU partition contains 16 EUs
- Each EU is a 5-issue SMT multithreaded superscalar (VLIW) pipelined processor
- Ideal exploitation: a 800 processor machine
- Internal EU operators include scalar arithmetic operations, as well as float operations: sin, cos, logarithm, sqrt, etc

RV870

- 20 EU partitions

Nvidia GPU: GeForce GTX - Fermi





- MIMD multiprocessor of 10 SIMT processors
- Each SIMT is a SIMD architecture with 3 - 16 {EU} partitions, 8 EUs (CUDA) per partition

GPU: programming model

- Current tools (e.g. CUDA) are too elementary and low-level
- Serious problems of programmability
  - Programmer is in charge of managing
    - data-parallelism at the architectural level
    - memory and I/O
    - multithreading
    - communication
    - load balancing
- Trend (?)
  - High level programming model (structured parallel programming ?) with structured and/or compiler-based cooperation between Host (possibly MIMD) and SIMD coprocessors.