

Computer Architecture

Lecture 28: Fine-Grained Multithreading

Prof. Onur Mutlu

ETH Zürich

Fall 2020

4 January 2021

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)

Recall: How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

How to Handle Control Dependences

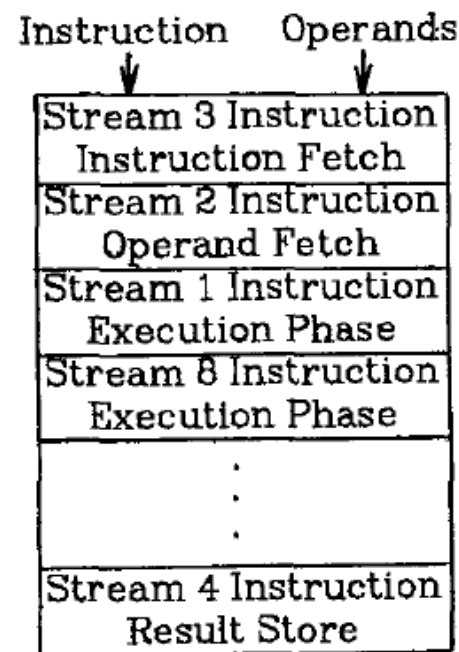
- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- **Potential solutions if the instruction is a control-flow instruction:**
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
 - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
 - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Fine-Grained Multithreading: History

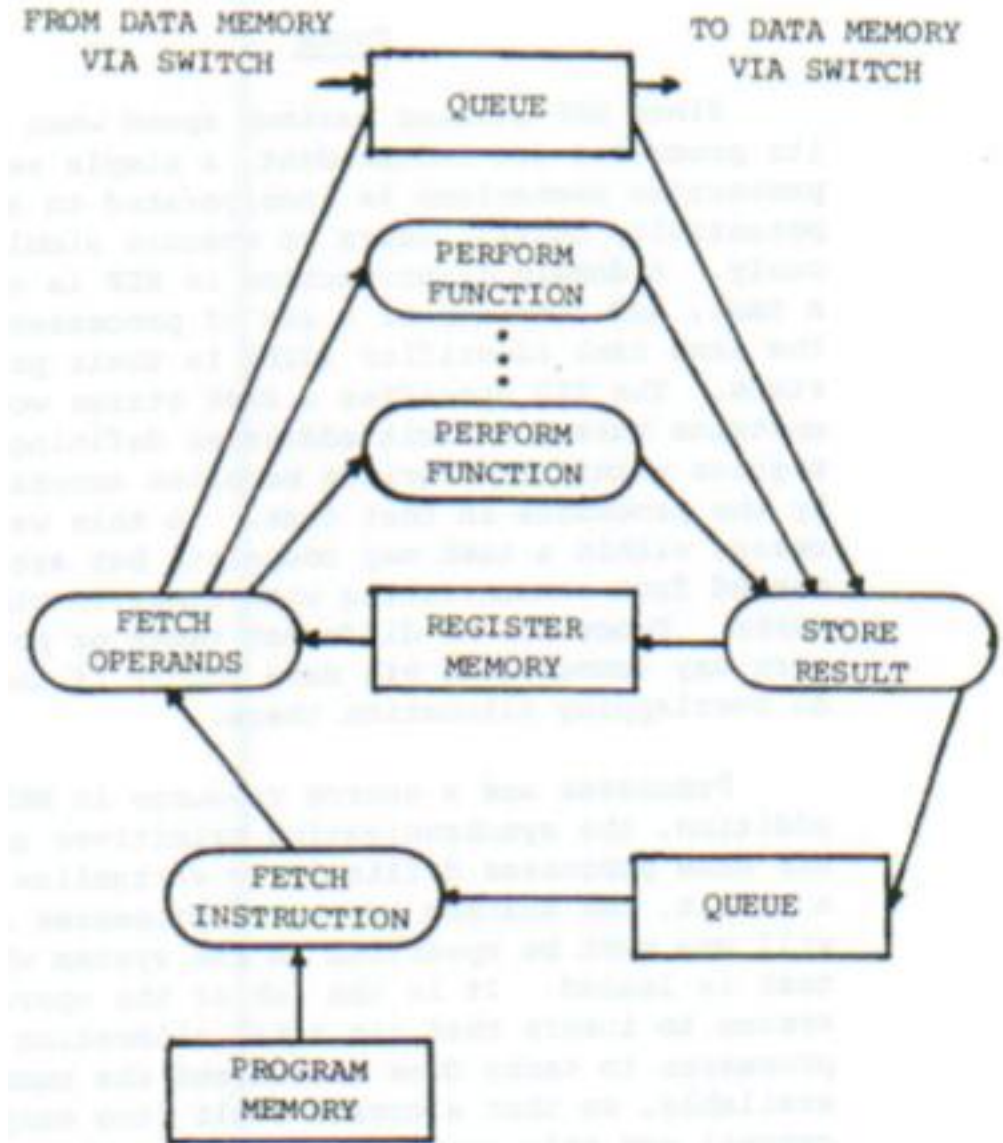
- CDC 6600's peripheral processing unit is fine-grained multithreaded
 - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
 - Processor executes a different I/O thread every cycle
 - An operation from the same thread is executed every 10 cycles
- Denelcor HEP (Heterogeneous Element Processor)
 - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
 - 120 threads/processor
 - available queue vs. unavailable (waiting) queue for threads
 - each thread can have only 1 instruction in the processor pipeline; each thread independent
 - to each thread, processor looks like a non-pipelined machine
 - system throughput vs. single thread performance tradeoff

Fine-Grained Multithreading in HEP

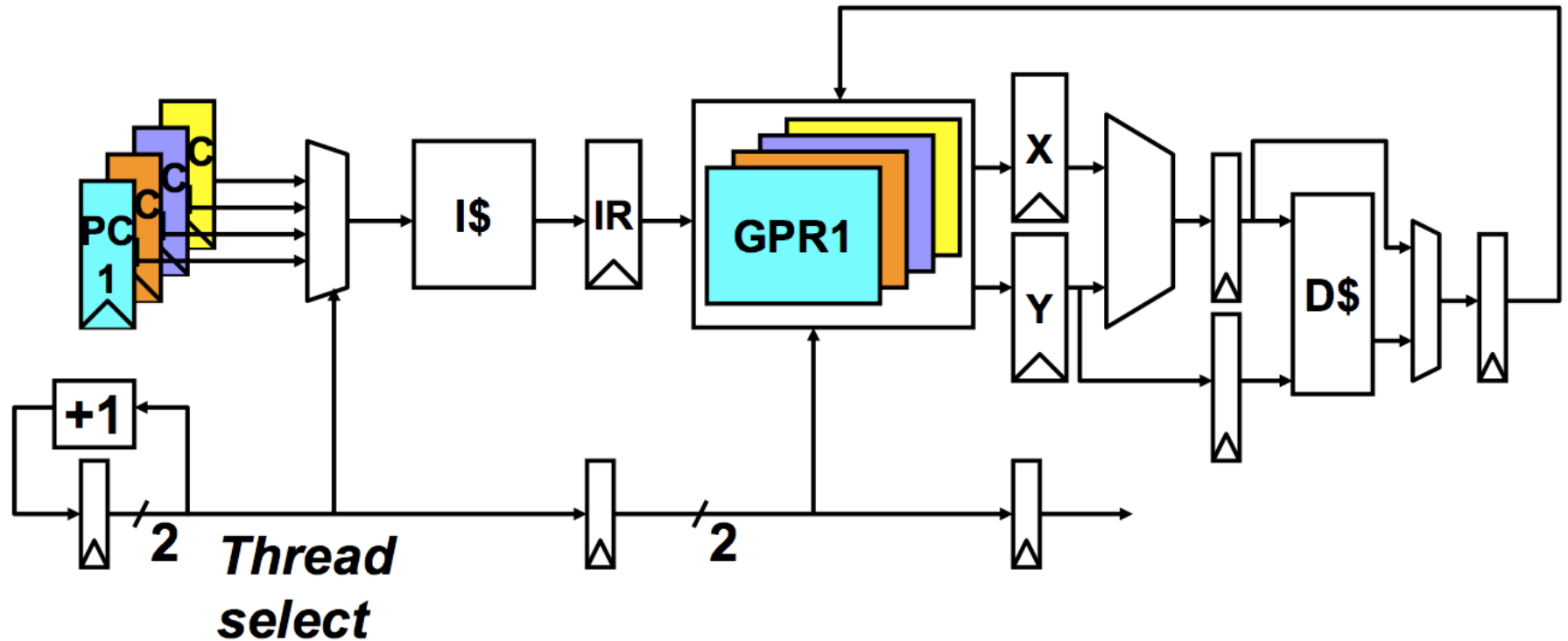
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
 - ❑ assuming no memory access
- No control and data dependency checking



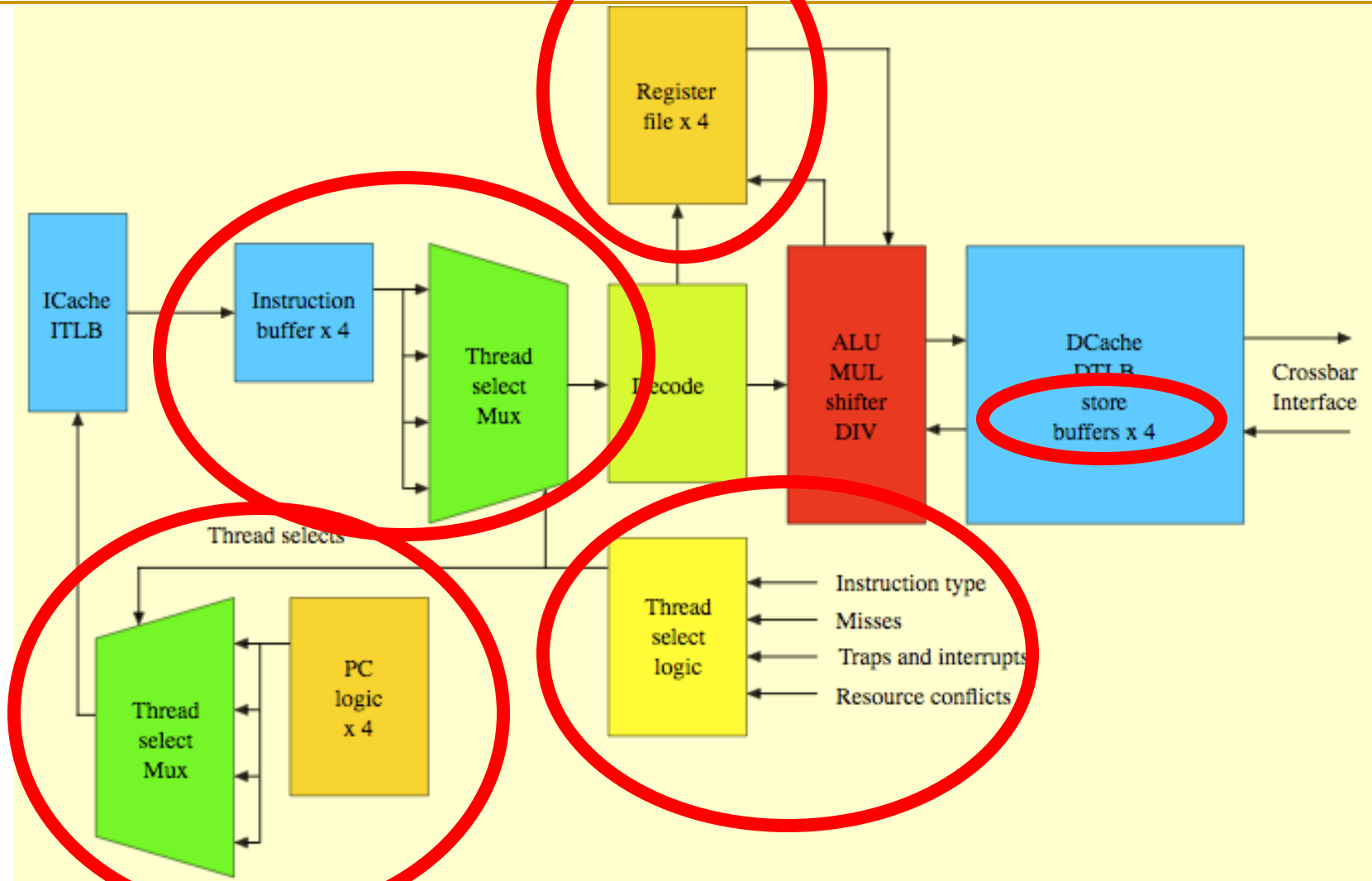
Burton Smith
(1941-2018)



Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-grained Multithreading

■ Advantages

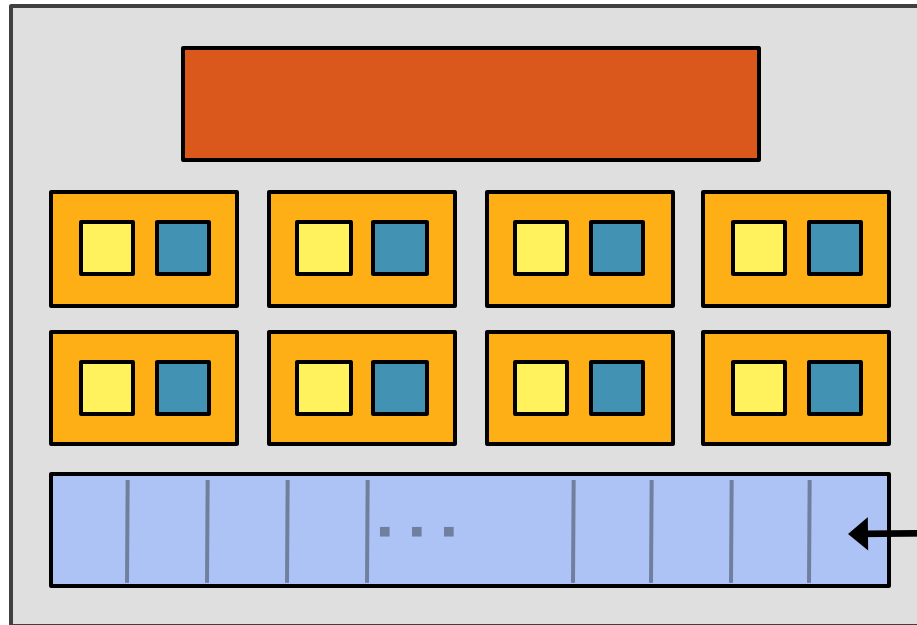
- + No need for dependency checking between instructions
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

■ Disadvantages

- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Some dependency checking logic *between* threads remains (load/store)

Modern GPUs are FGMT Machines

NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)



= data-parallel (SIMD) func. unit,
control shared across 8 units

■ = multiply-add
■ = multiply

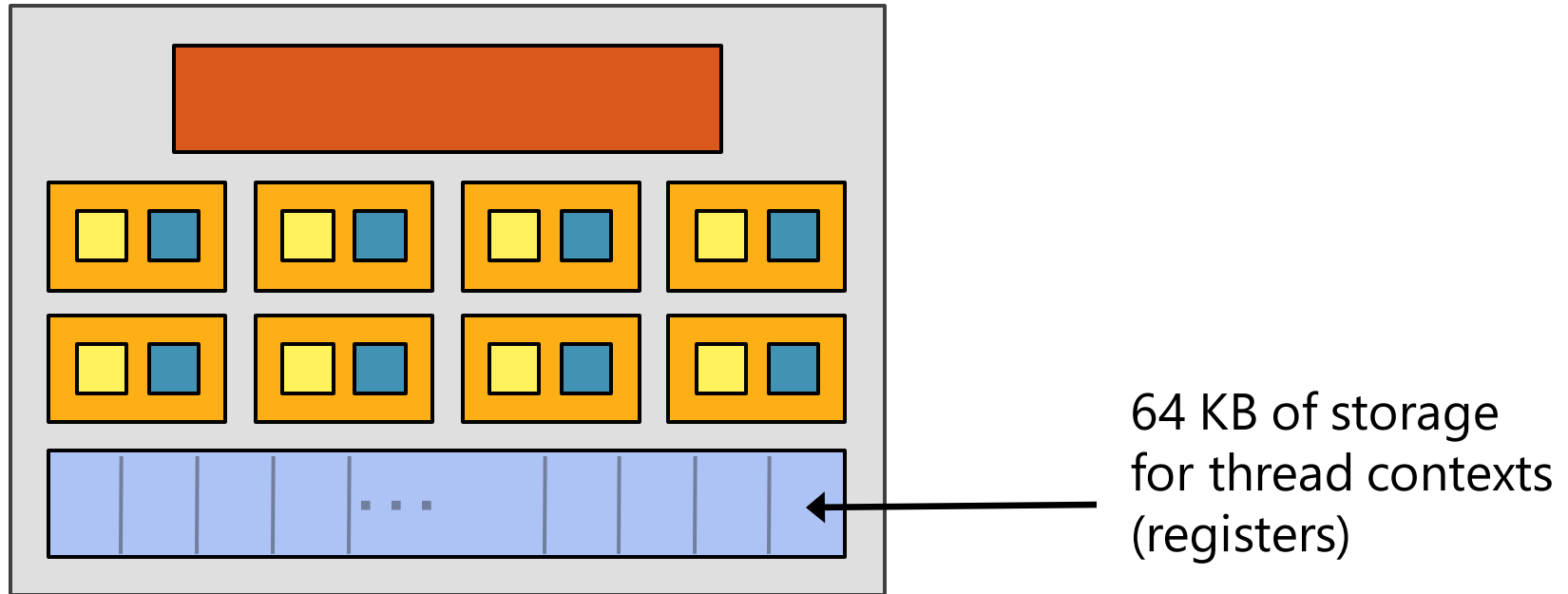


= instruction stream decode



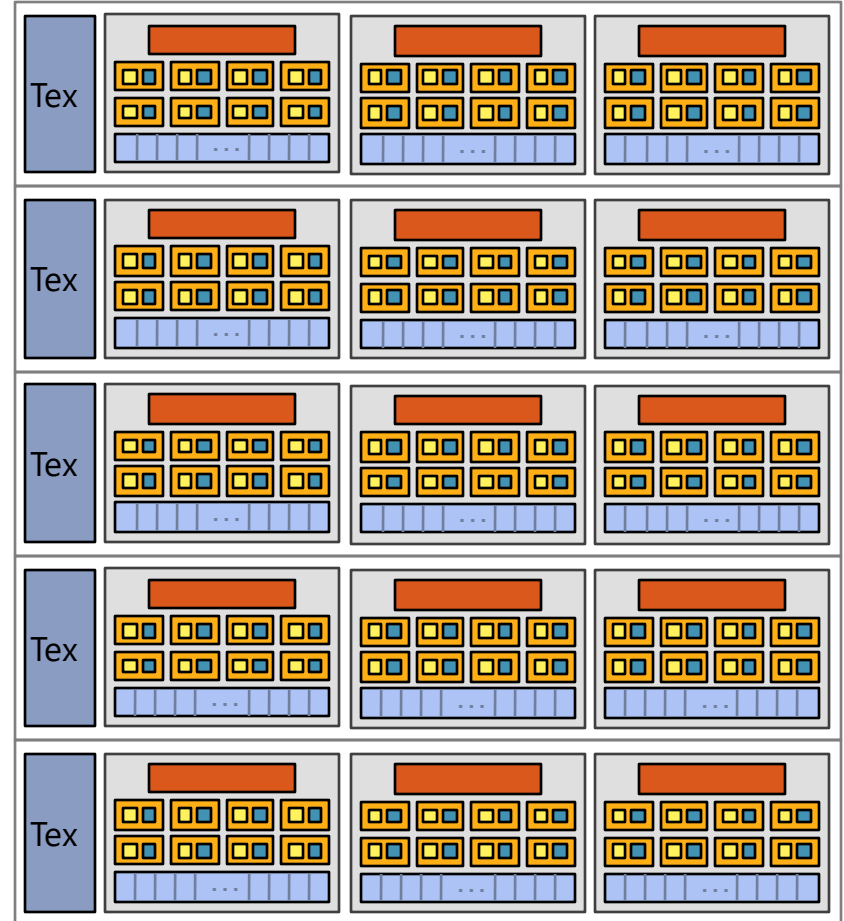
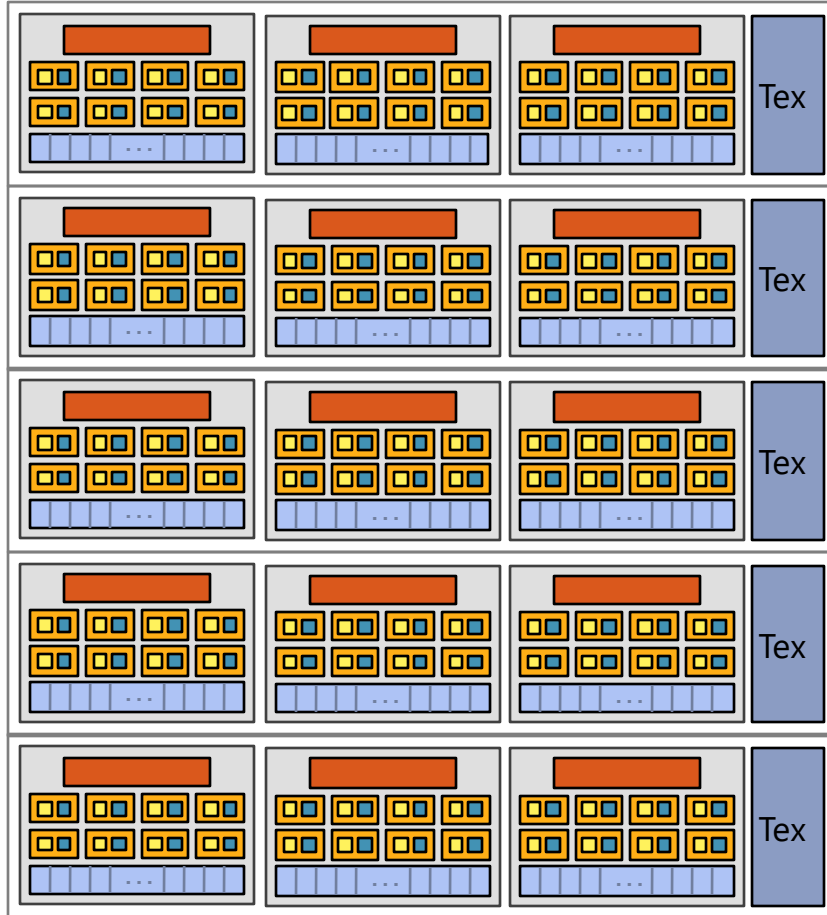
= execution context storage

NVIDIA GeForce GTX 285 “core”



- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

End of Fine-Grained Multithreading

In Memory of Burton Smith

A PIPELINED, SHARED RESOURCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205



Burton Smith
(1941-2018)

Architecture and applications of the HEP multiprocessor computer system

Burton J. Smith
Denelcor, Inc., 14221 E. 4th Avenue, Aurora, Colorado 80011

In Memory of Burton Smith (II)

The Tera Computer System*

Robert Alverson

David Callahan
Allan Porterfield

Daniel Cummings
Burton Smith

Brian Koblenz

Tera Computer Company
Seattle, Washington USA

4 Processors

Each processor in a Tera computer can execute multiple instruction streams simultaneously. In the current implementation, as few as one or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors. When an instruction finishes, the stream to which it belongs thereby becomes ready to execute the next instruction. As long as there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is being fully utilized. Thus, it is only necessary to have enough streams to hide the expected latency (perhaps 70 ticks on average); once latency is hidden the processor is running at peak performance and additional streams do not speed the result.

Computer Architecture

Lecture 28: Fine-Grained Multithreading

Prof. Onur Mutlu

ETH Zürich

Fall 2020

4 January 2021