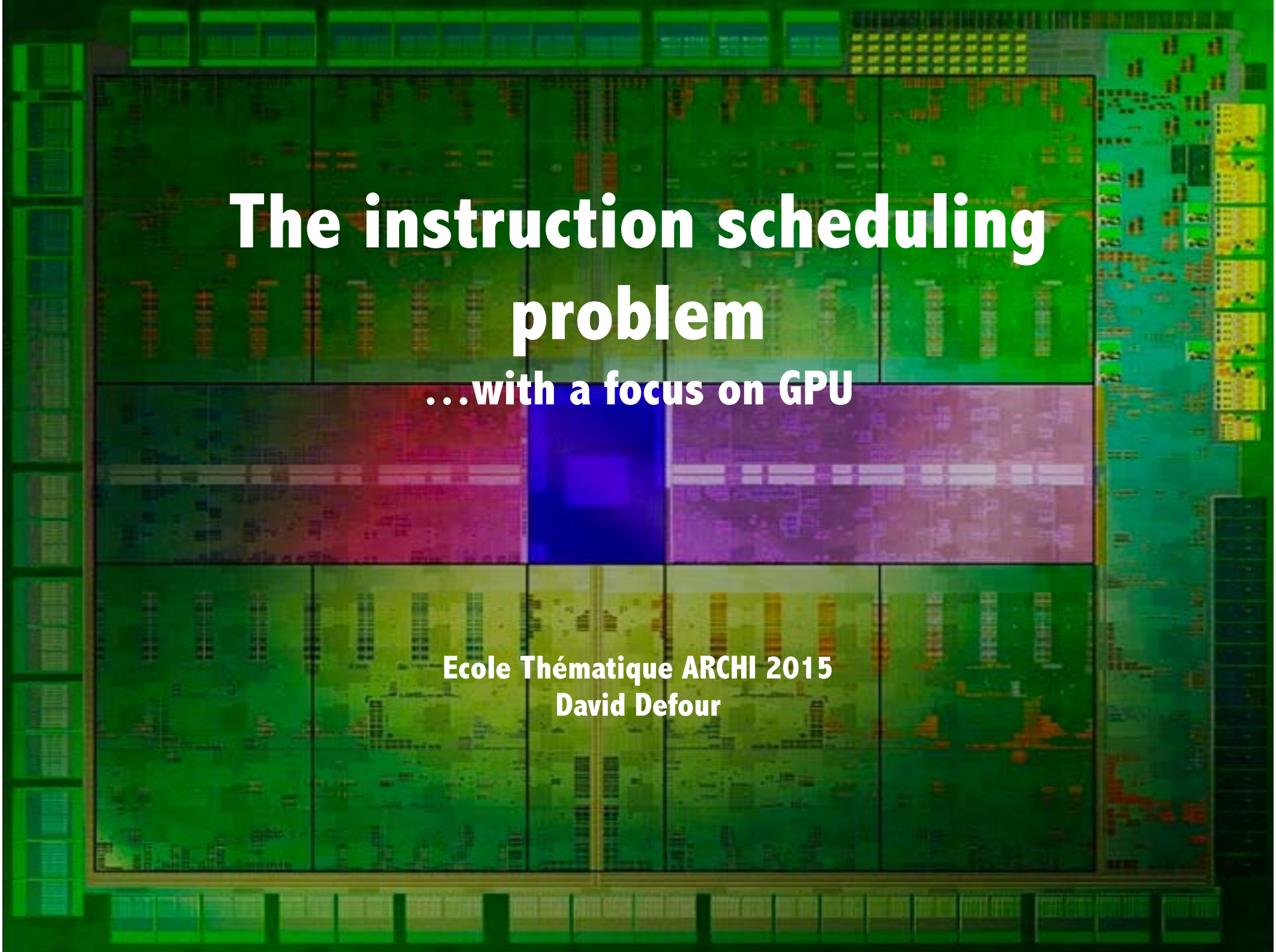


The instruction scheduling problem

...with a focus on GPU

Ecole Thématique ARCHI 2015
David Defour

Scheduling in GPU's

- Stream are scheduled among GPUs
- Kernel of a Stream are scheduler on a given GPUs using **FIFO**
- Block are scheduled among SMs with a **round robin** algorithm
- Warp are scheduled to SIMT processor using a **round robin** algorithm with **priority** and **aging**
- Data dependencies are handle using a **scoreboard**

Topics covered

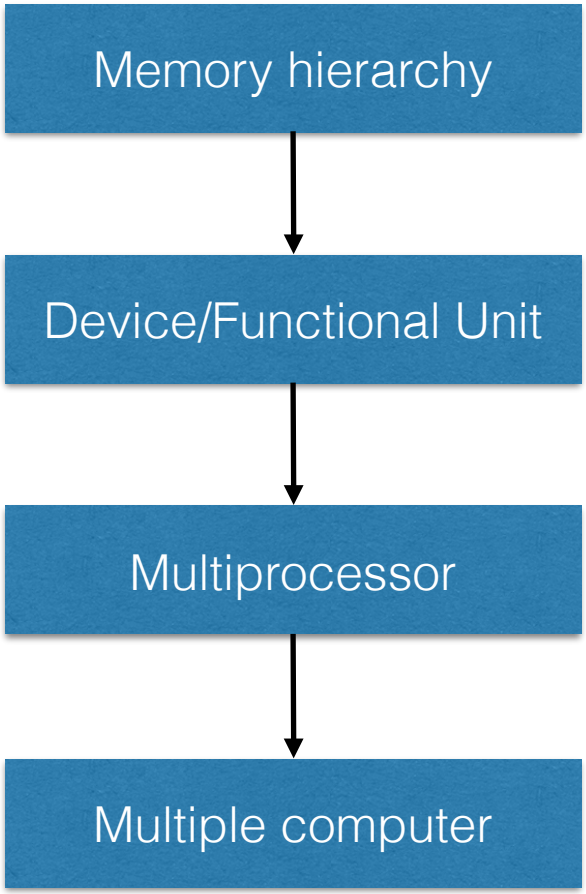
- **Definition of the scheduling problem**
- Scheduling on superscalar architecture
- Scheduling on vector architecture
- How to schedule
- New problems

Principle of scheduling

- Complexity of Networks, Computers, Processors & Applications is increasing
- Need of efficient scheduling at each level

Level of Scheduling

- Intra-computer
  - Local scheduling (memory hierarchy, device, functional unit…)
- Computer level
  - Among cores (threads)
- Inter-computer
  - Global scheduling (tasks)



The scheduling problem: the 5 components

1. Events
   Smallest indivisible schedulable entity

Set 1
(tasks, job, inst…)

Set 2
(processors, core, FU…)

The scheduling problem: the 5 components

Processing power

Interconnect

Processor

1. Events
   Smallest indivisible schedulable entity
2. Environment
   Characteristics and capabilities of the surrounding that will impact schedule

Set 1
(tasks, job, inst…)

Set 2
(processors, core, FU…)

The scheduling problem: the 5 components

Processing power

Interconnect

Processor

1. Events
   Smallest indivisible schedulable entity
2. Environment
   Characteristics and capabilities of the surrounding that will impact schedule
3. Requirement
   Real time deadline, improvement in performance

Set 1
(tasks, job, inst…)

Set 2
(processors, core, FU…)

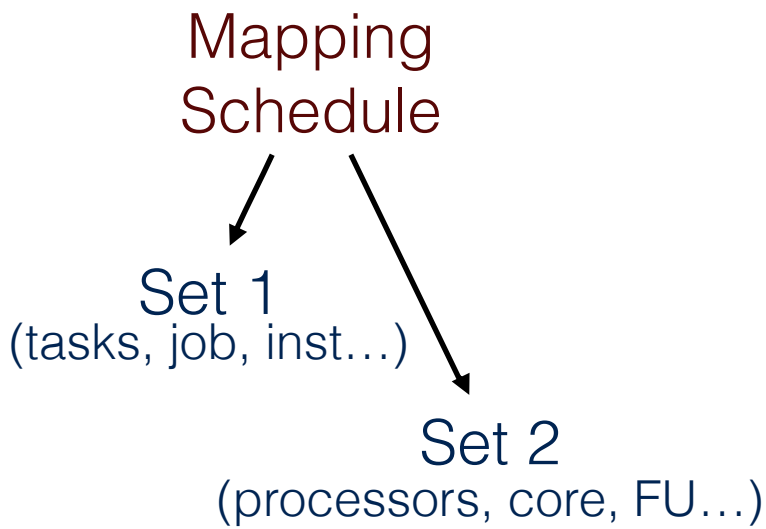
Objectives
(performance, power consumption, ..)

The scheduling problem: the 5 components

Processing power

Interconnect

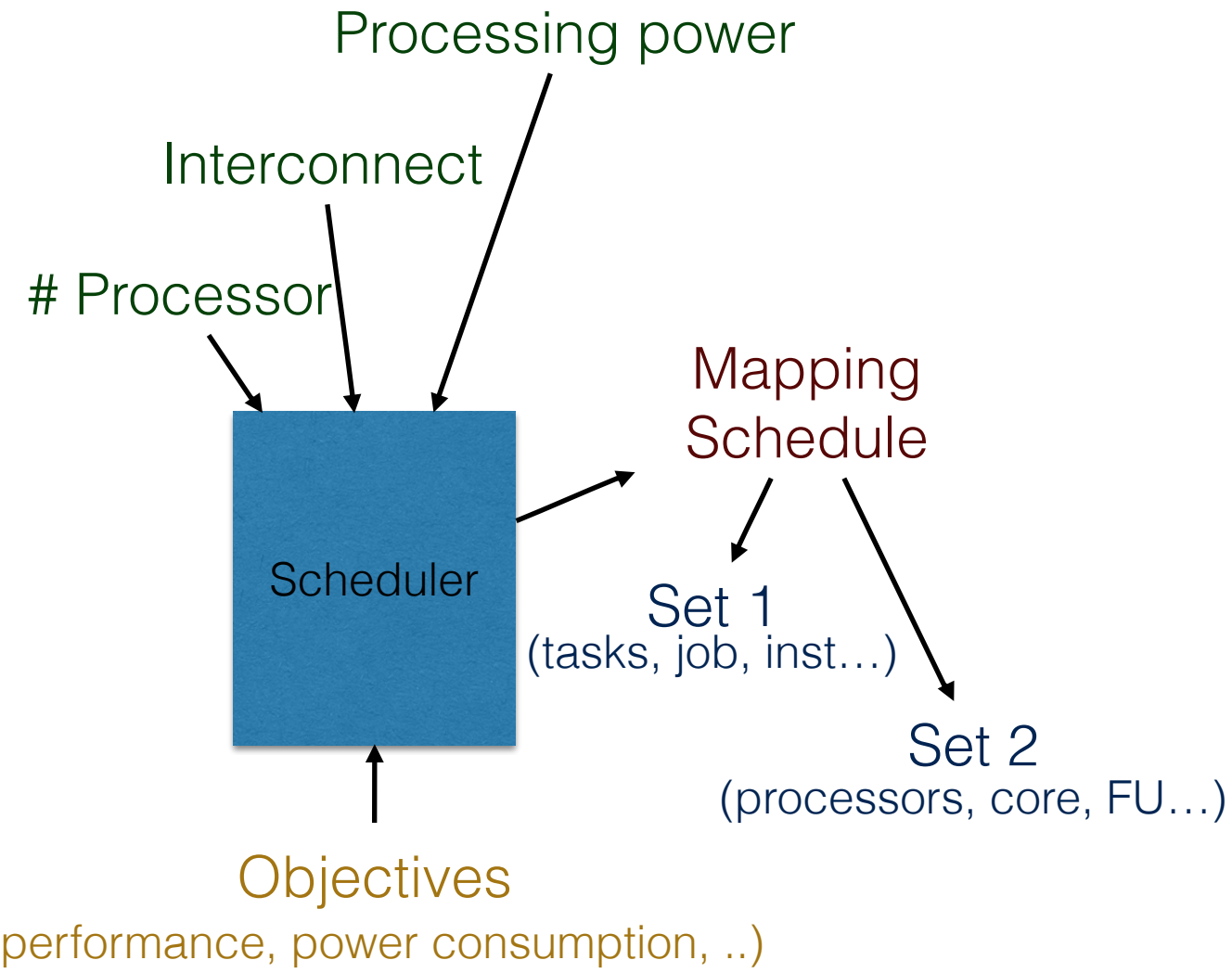
Processor



Objectives
(performance, power consumption, ..)

1. Events
   Smallest indivisible schedulable entity
2. Environment
   Characteristics and capabilities of the surrounding that will impact schedule
3. Requirement
   Real time deadline, improvement in performance
4. Schedule
   Set of ordered pairs of events and times

The scheduling problem: the 5 components



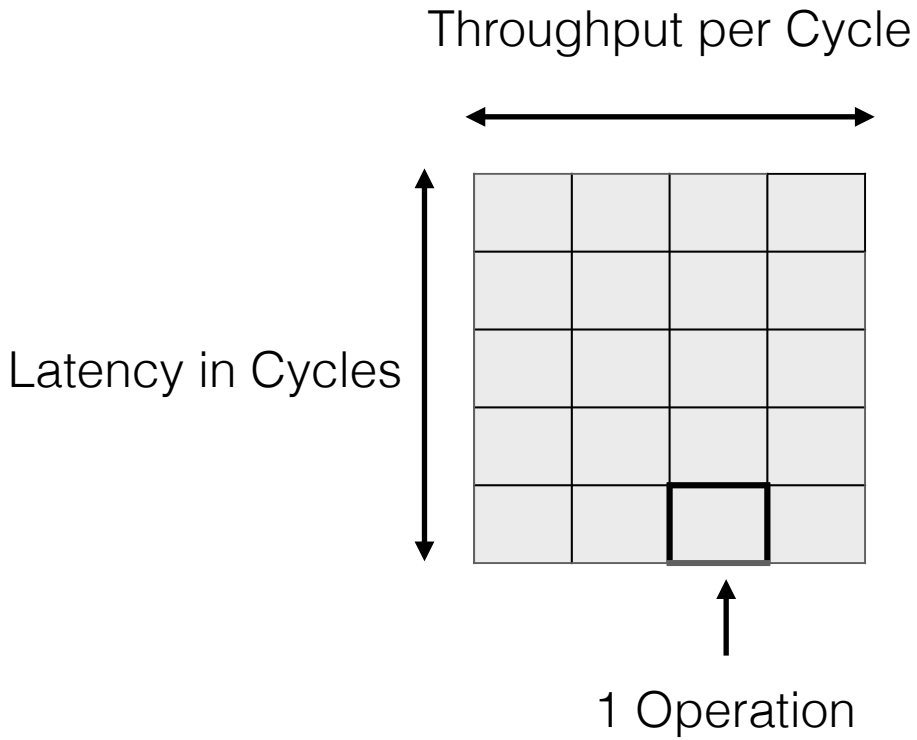
1. Events
   Smallest indivisible schedulable entity
2. Environment
   Characteristics and capabilities of the surrounding that will impact schedule
3. Requirement
   Real time deadline, improvement in performance
4. Schedule
   Set of ordered pairs of events and times
5. Scheduler
   Takes events, Environment & requirement and produce a schedule

Topics covered

- Definition of the scheduling problem
- **Scheduling on superscalar architecture**
- Scheduling on vector architecture
- How to schedule
- New problems

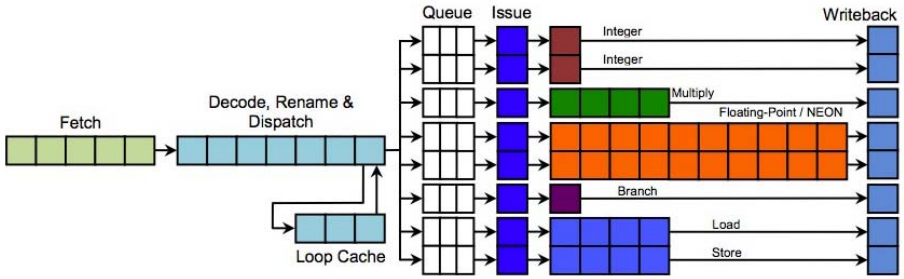
Scheduling on superscalar architecture

- Use available transistors to do more work per unit of time
  - Issue multiple instruction per clock cycle, deeper pipeline to increase ILP
  - Parallelism = Troughput * Latency



Instruction Level Parallelism (ILP)

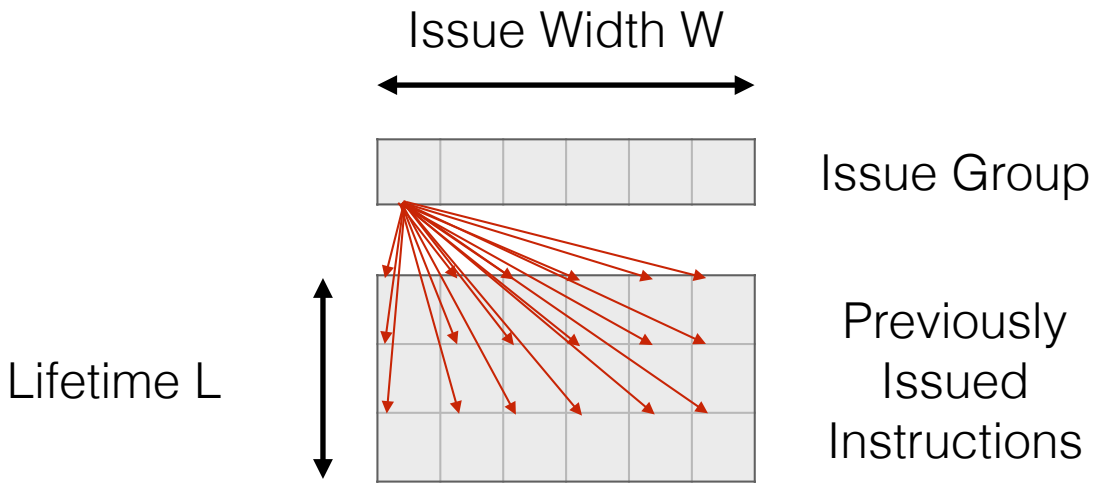
- Increase in instruction width and depth of machine pipeline
  - Increase of independent instruction required to keep processor busy
  - Increase the number of partially in flight executed instructions
- Difficult to build machines that control large numbers of in-flight instructions
  - Either for dynamically or statically schedule machine



Dynamically scheduled machine

- Growth of
  - Instruction windows
  - Reorder buffer
  - Rename Register files
- Number of port on each of those element must grow with the issue width
- The logic to track dependencies in-flight instruction grow cubically in the number of instructions

Control logic scaling



- Each issued inst. must make interlock checks against W*L inst., Growth in interlock ~W*(W*L)
  - In-order machines: L related to pipeline latency
  - Out-of-order machines: L related to inst. buffer (ROB, …)
- Increase in W
  - => increase in inst. windows to find enough //
  - => greater L
  - => Out-Of-Order logic grows in ~W^3

Statically scheduled machine (VLIW)

- Shift most of the schedule burden to the compiler
  - To support more in-flight instructions, requires more registers, more ports per register, more hazard interlock logic
  - Results in a quadratic increase of complexity and circuit size

Topics covered

- Definition of the scheduling problem
- Scheduling on superscalar architecture
- **Scheduling on vector architecture**
- How to schedule
- New problems

Vector architecture

- Basic idea:
  - Read set of data elements into « vector registers »
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector processing (1)

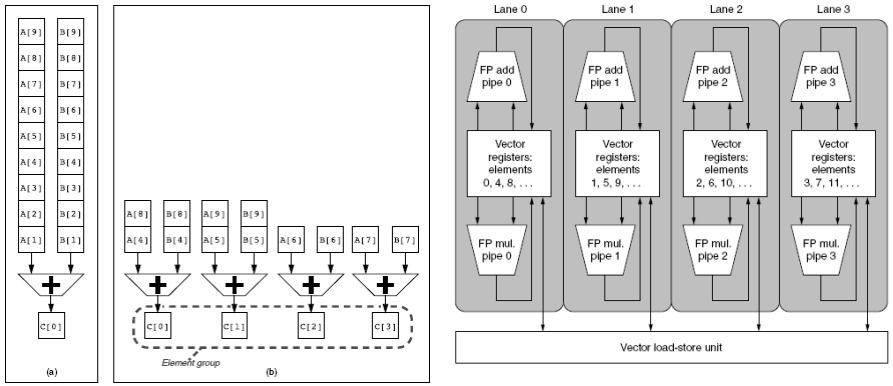
- Vector processing solves some problems related to ILP logic

  1. A single instruction specifies a great deal of work
     (Execute an entire loop)

     - Reduce instruction fetch and decode bandwidth needed to keep multiple deeply pipeline FU busy

  2. Compiler/programmer indicates that the computation of each results in a vector is independent of the computation of other results in the same vector

     - Elements in a vector can be computed using an array of parallel FU or single very deeply pipeline FU or any intermediate configuration



Vector processing (2)

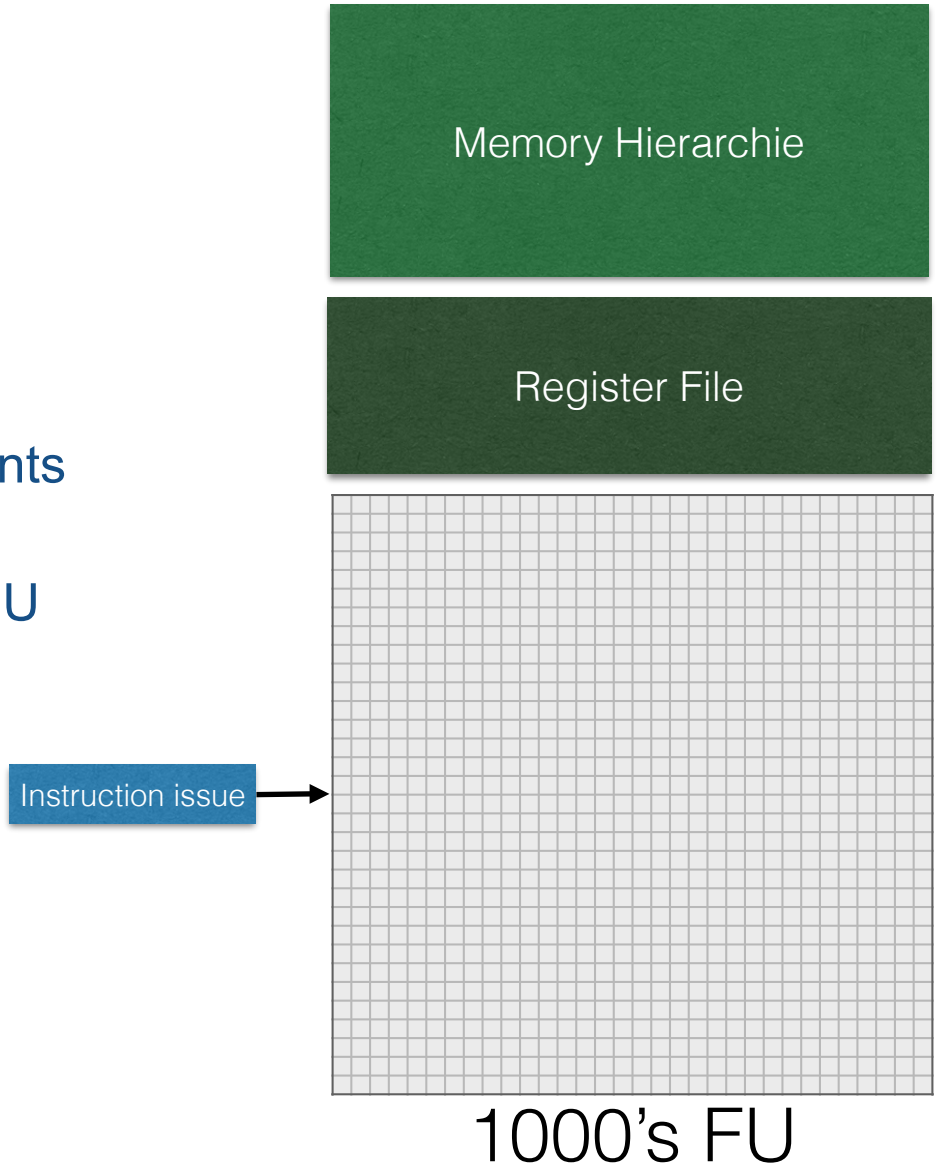
3. Hardware need only check for data hazards between two instructions once per vector operand, not for every element in a vector.
    - Reduce control logic
4. Vector instructions that access memory have known access pattern
    - Adjacent data within a vector can be fetch from a set of heavily interleaved memory banks
    - Access latency to main memory is seen only once, not for every element
5. Loops are replaced by vector instruction whose behavior is predetermined, control hazard that would arise from loop branch are non-existent
6. Traditional cache are replaced by prefetching and blocking technics to increase locality

Short vector: SIMD

- x86 processors:
  - Expect 2 additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD
- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers
  - Operands must be consecutive and aligned memory locations.

Long vector

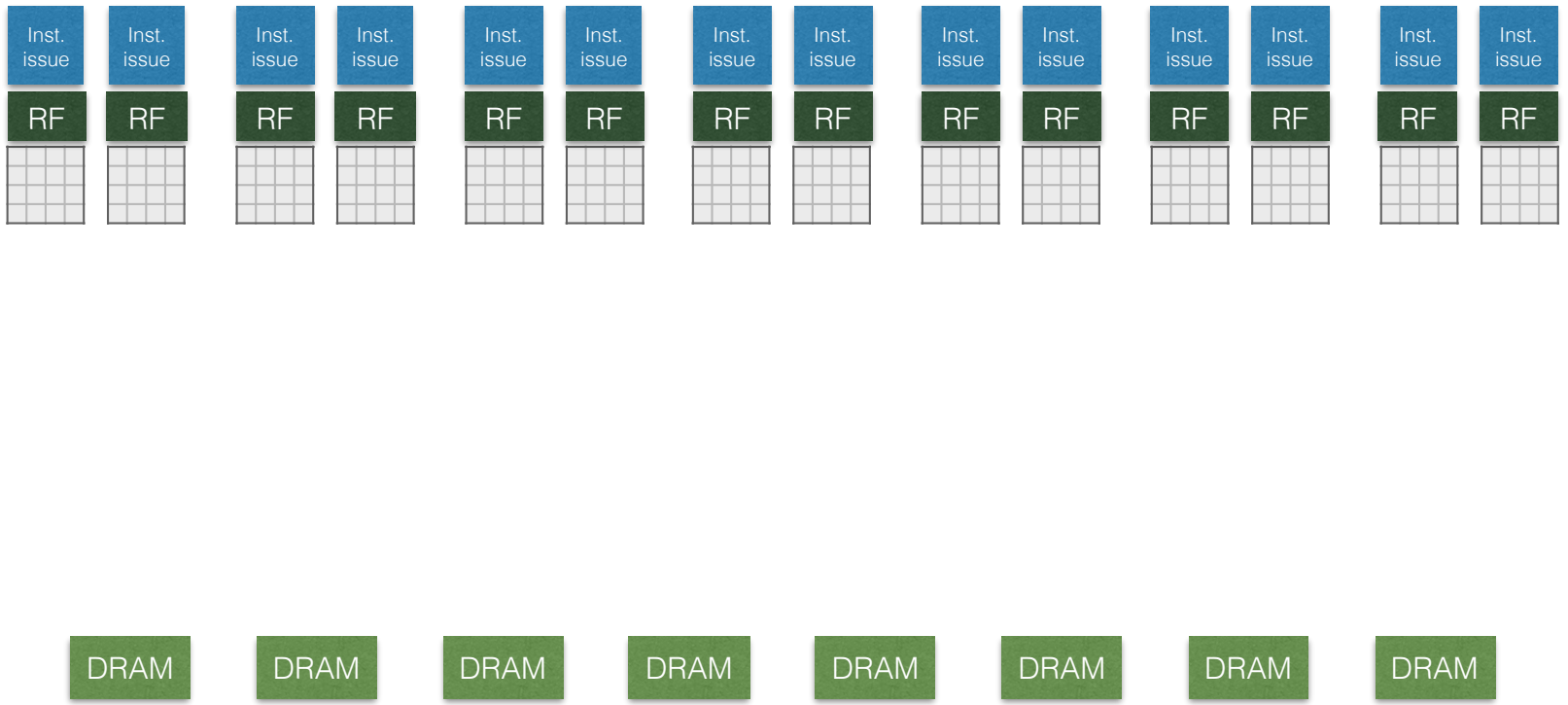
- Today's die are big enough to embed thousand's of FU
  - Problem:
    - Managing vector of 1000's elements
    - Issuing instructions over 1000's FU
    - Memory hierarchy accesses,
    - …
- Solution: Clustering



Long vector: Clustering

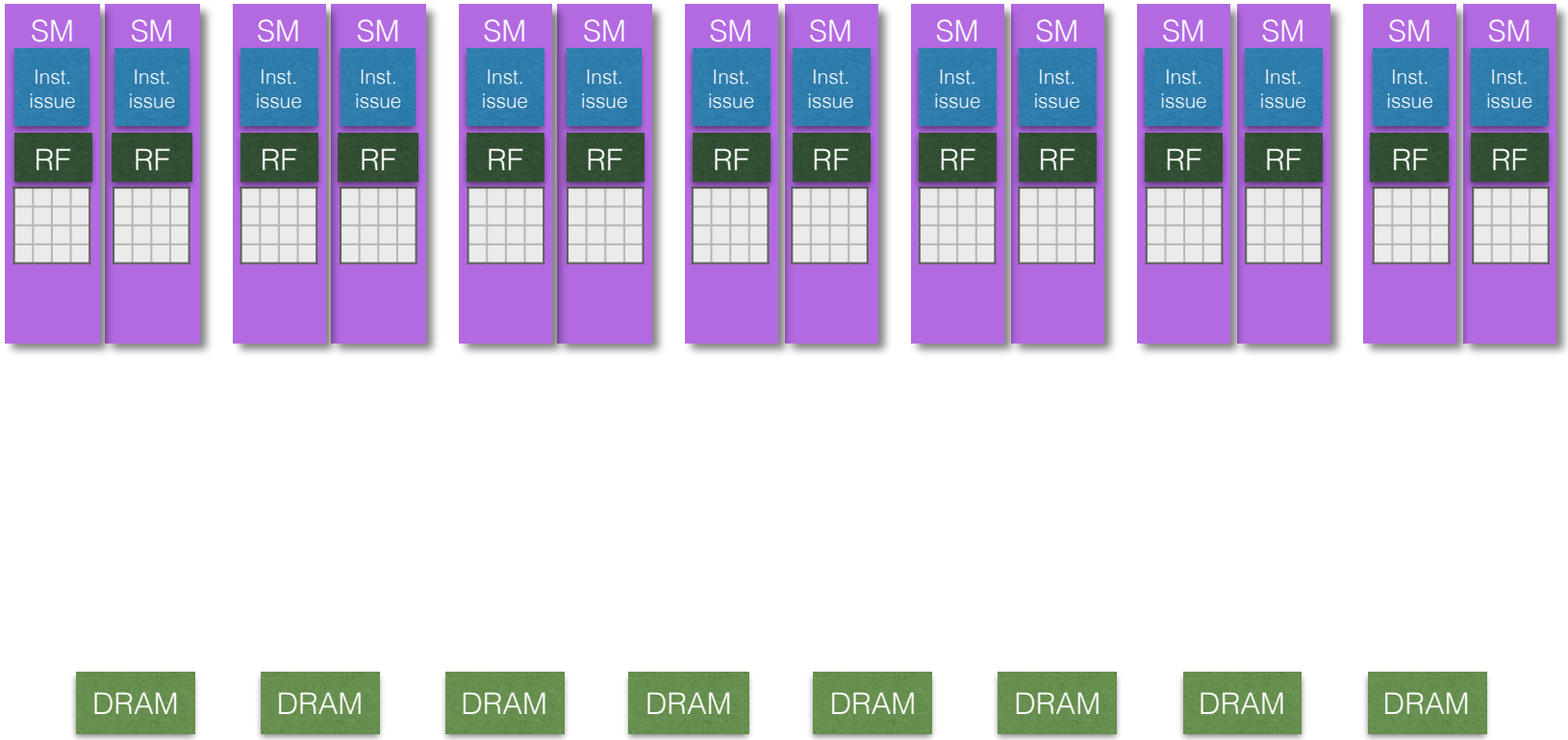
- Divide machine into cluster of local register files and local functional units
- Lower bandwidth / Higher latency interconnect between clusters
- Software responsible of correct mapping of computations and minimization of communication overhead

Long vector: Clustering



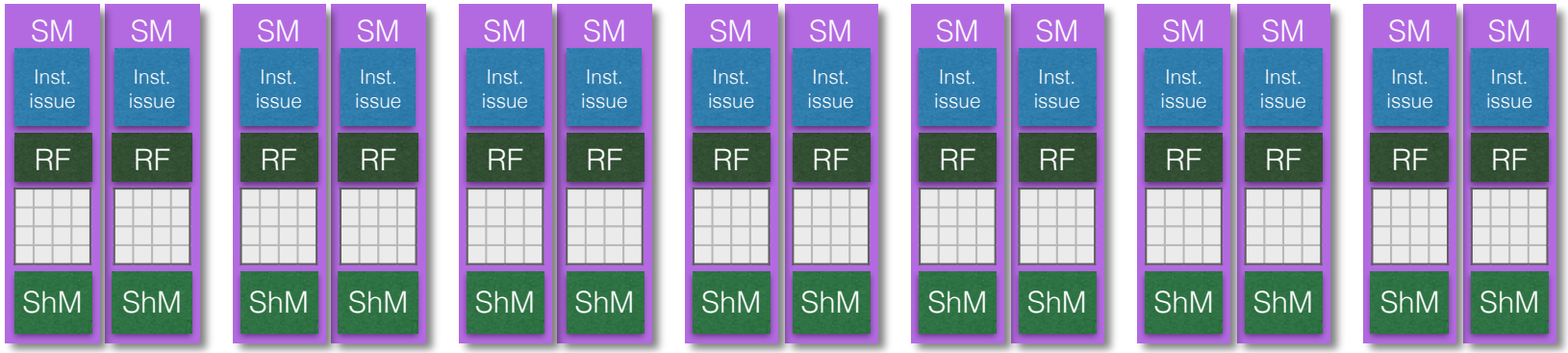
G80

Long vector: Clustering



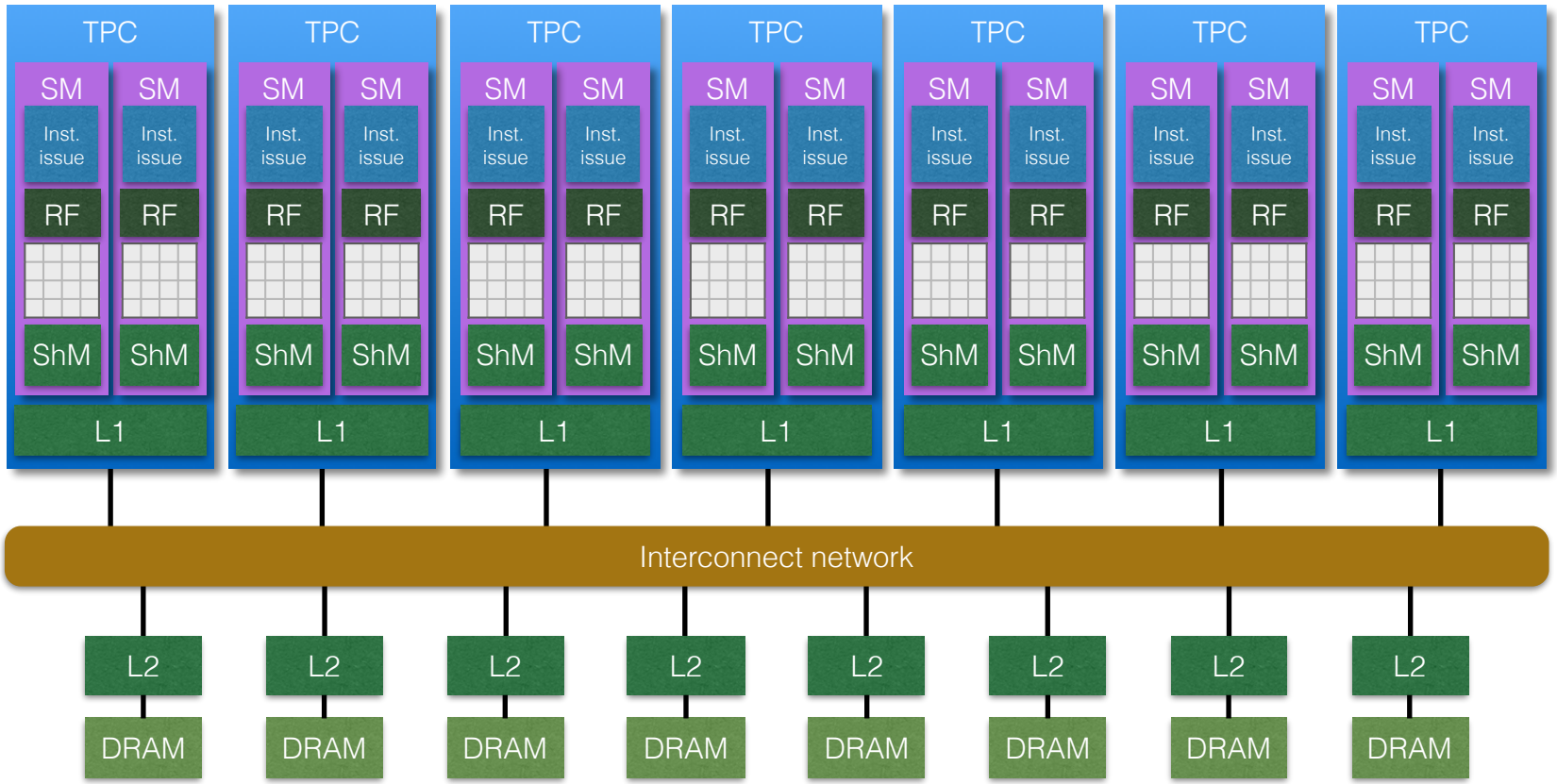
G80

Long vector: Clustering

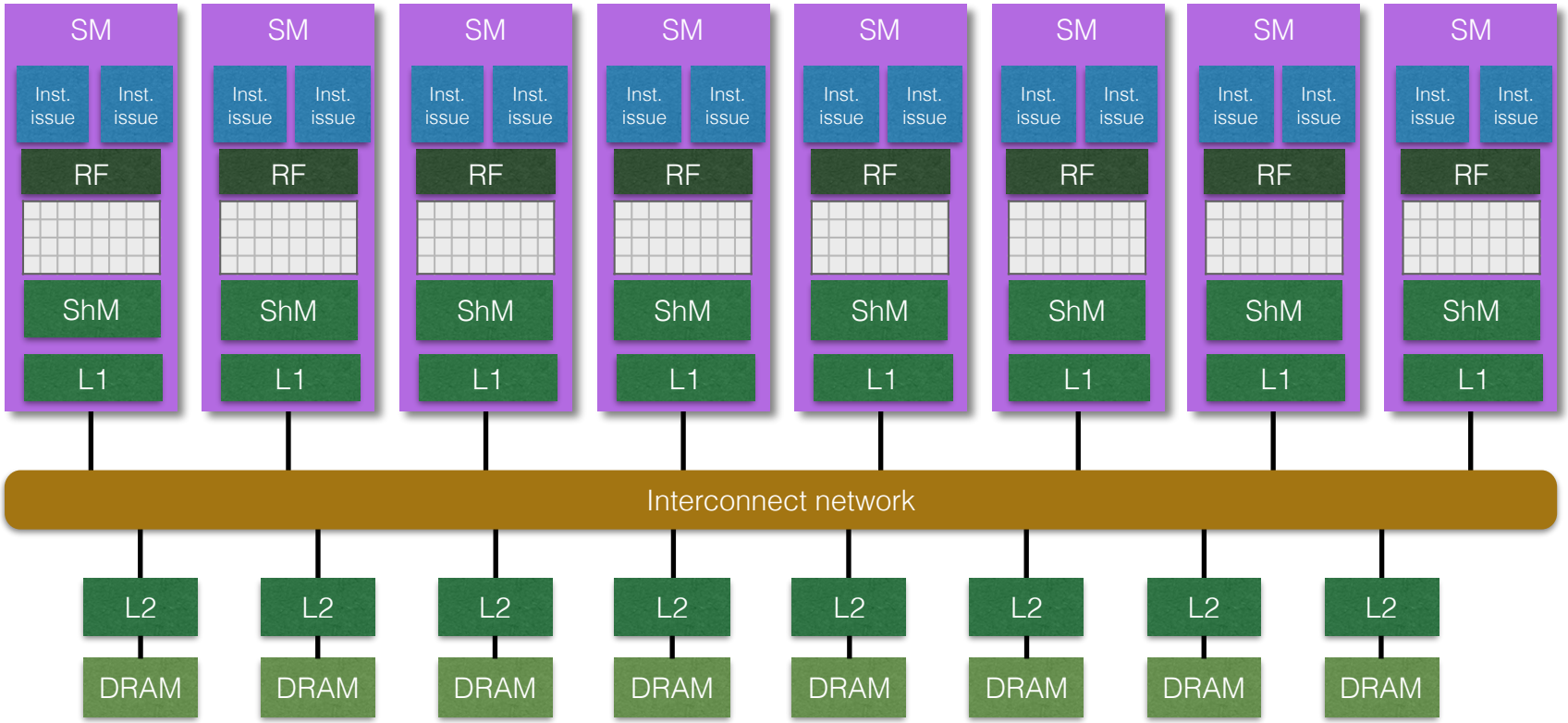


G80

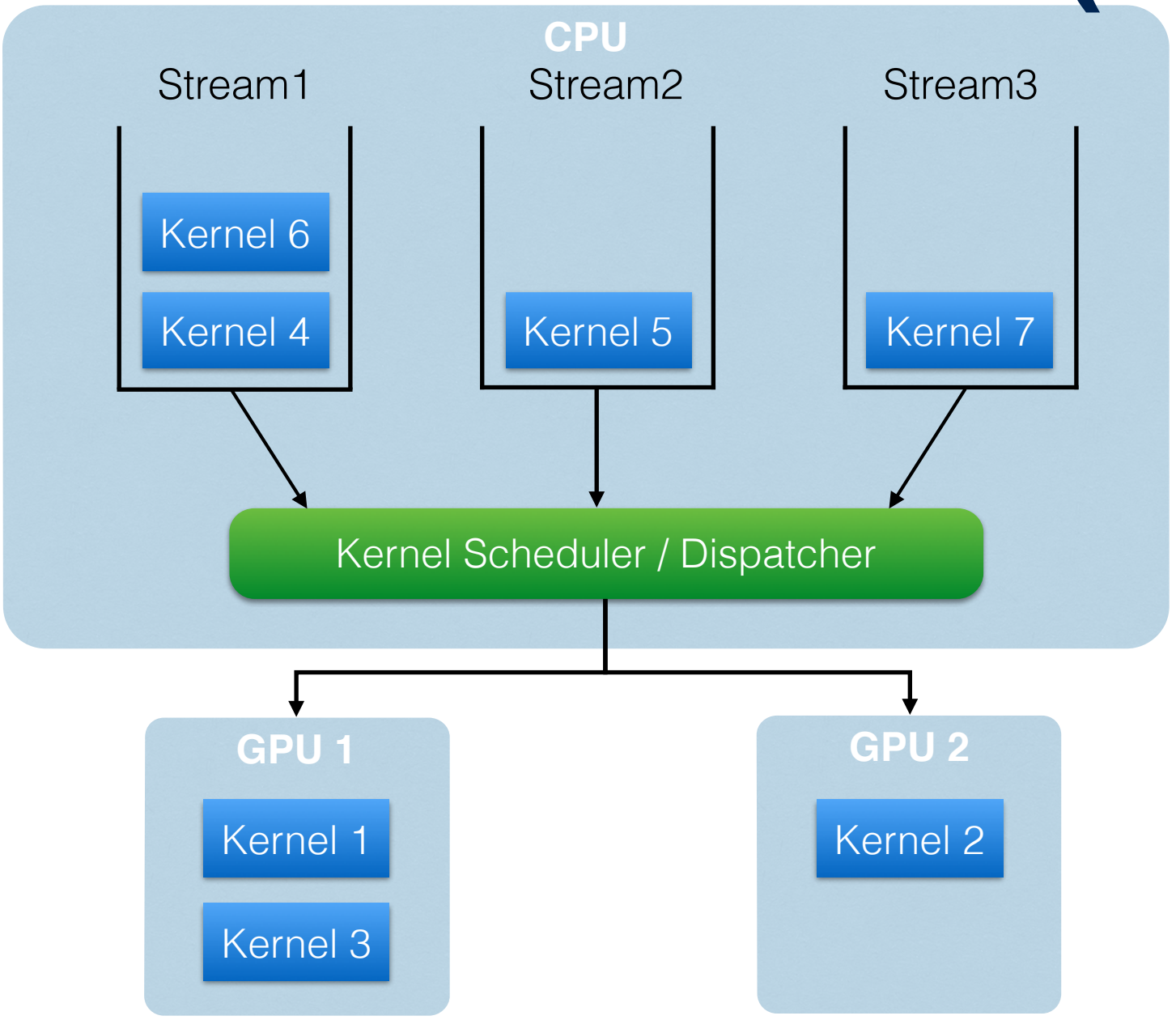
Long vector: Nvidia G80



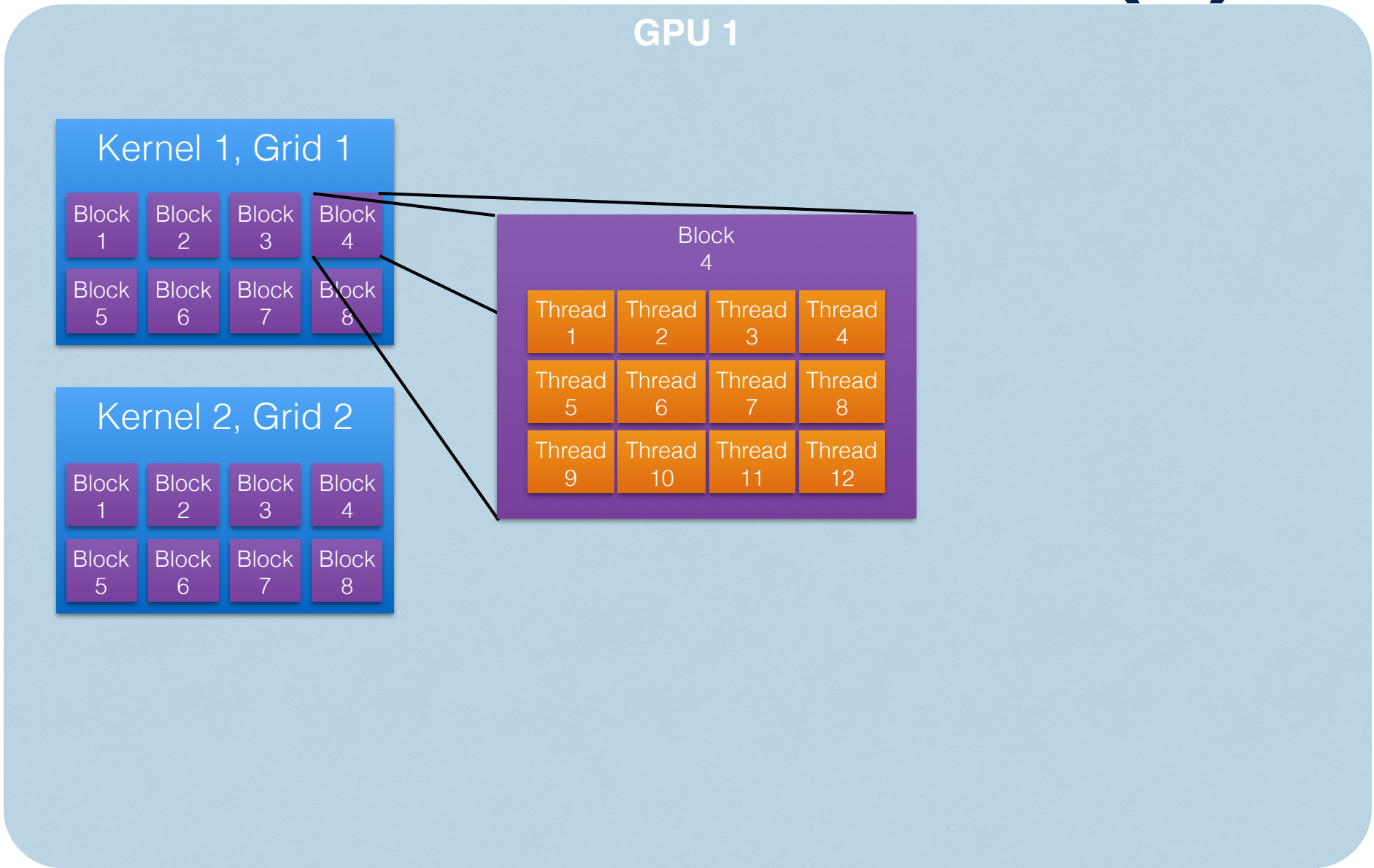
Long vector: FERMI



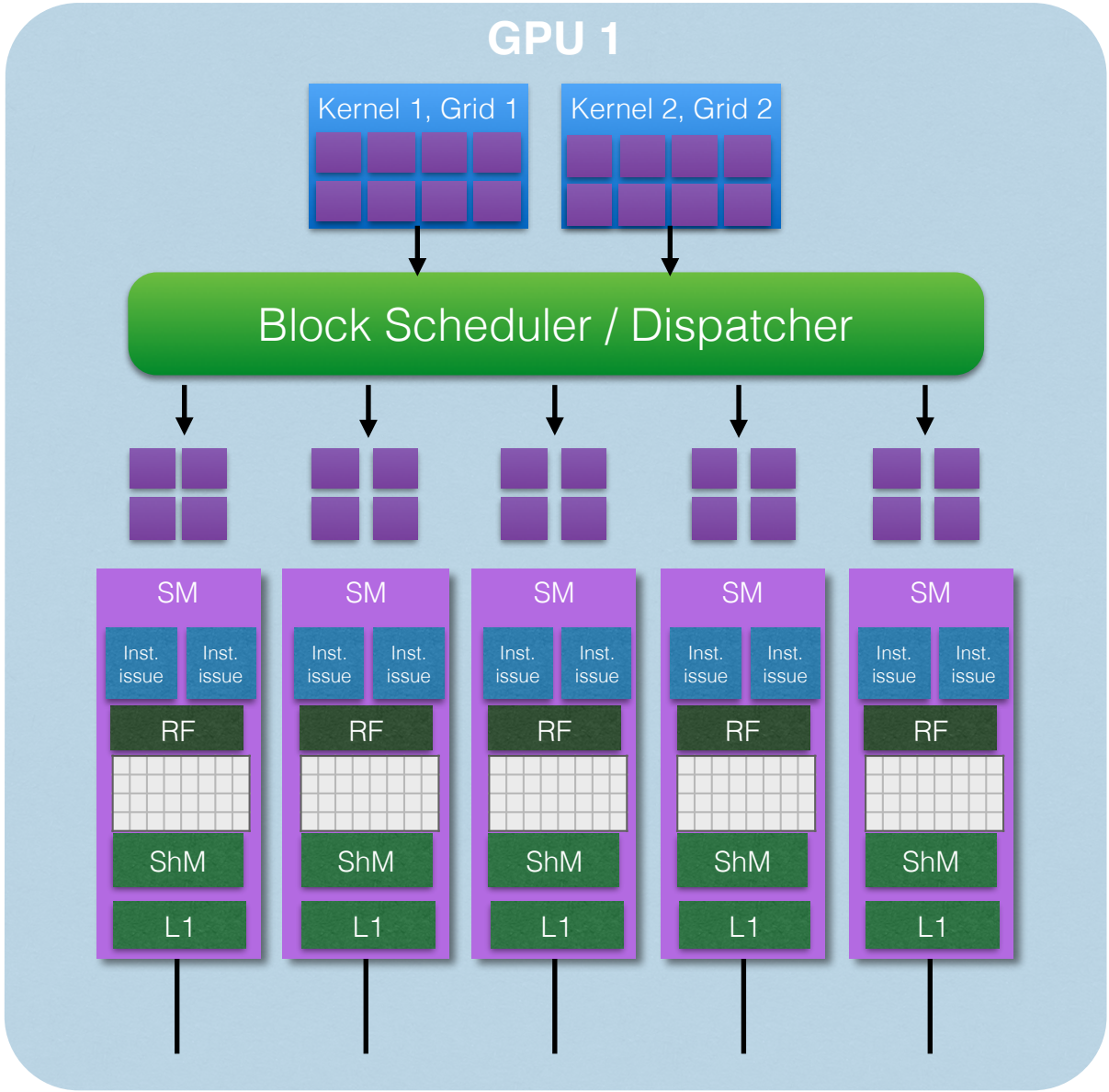
CUDA execution (1)



CUDA execution (2)



CUDA execution (3)



Kernel context state:

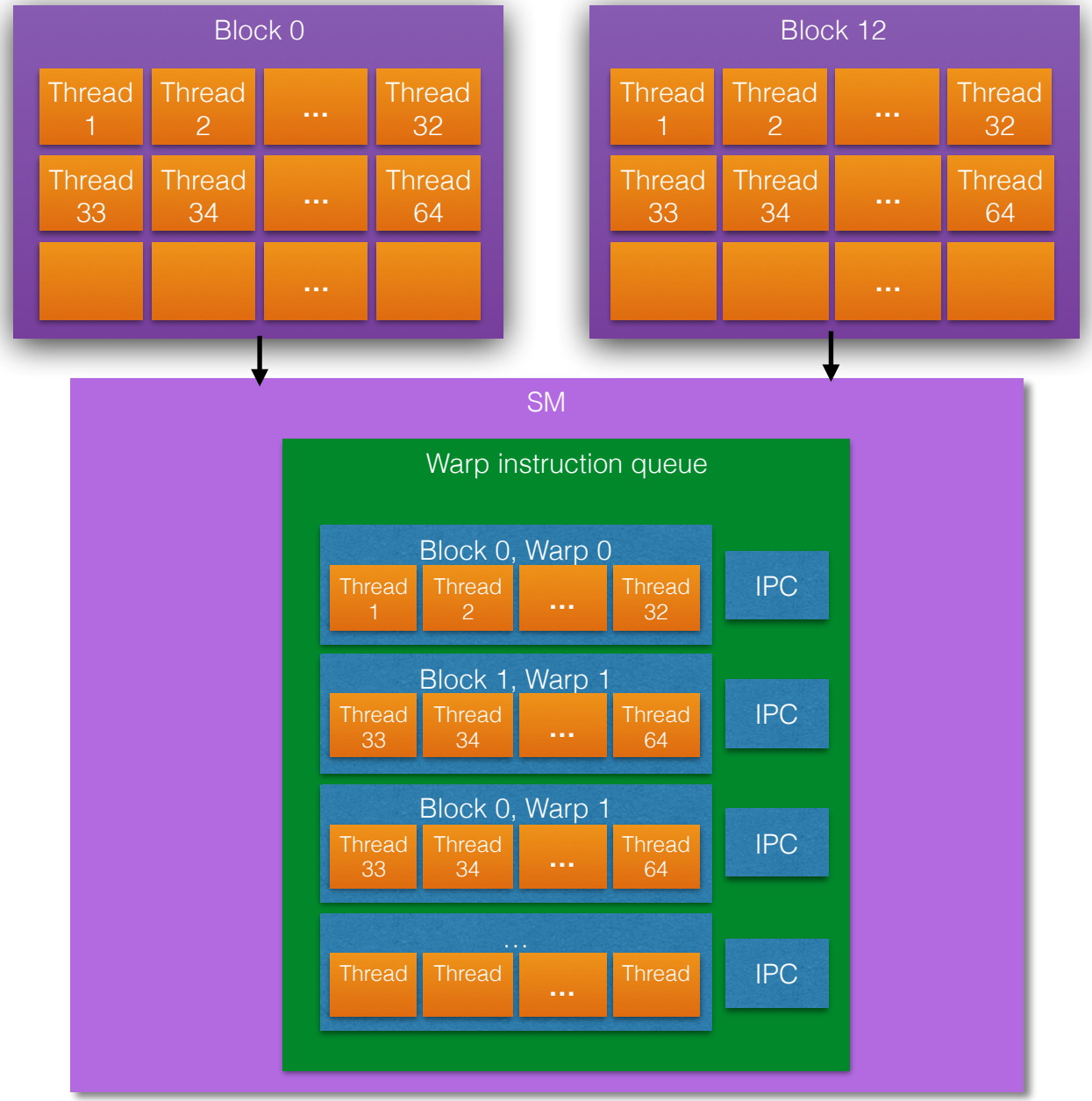
- TLB, dirty data in caches
- Registers, shared mem.

Fermi:

- Can dispatch 1 kernel to each SM

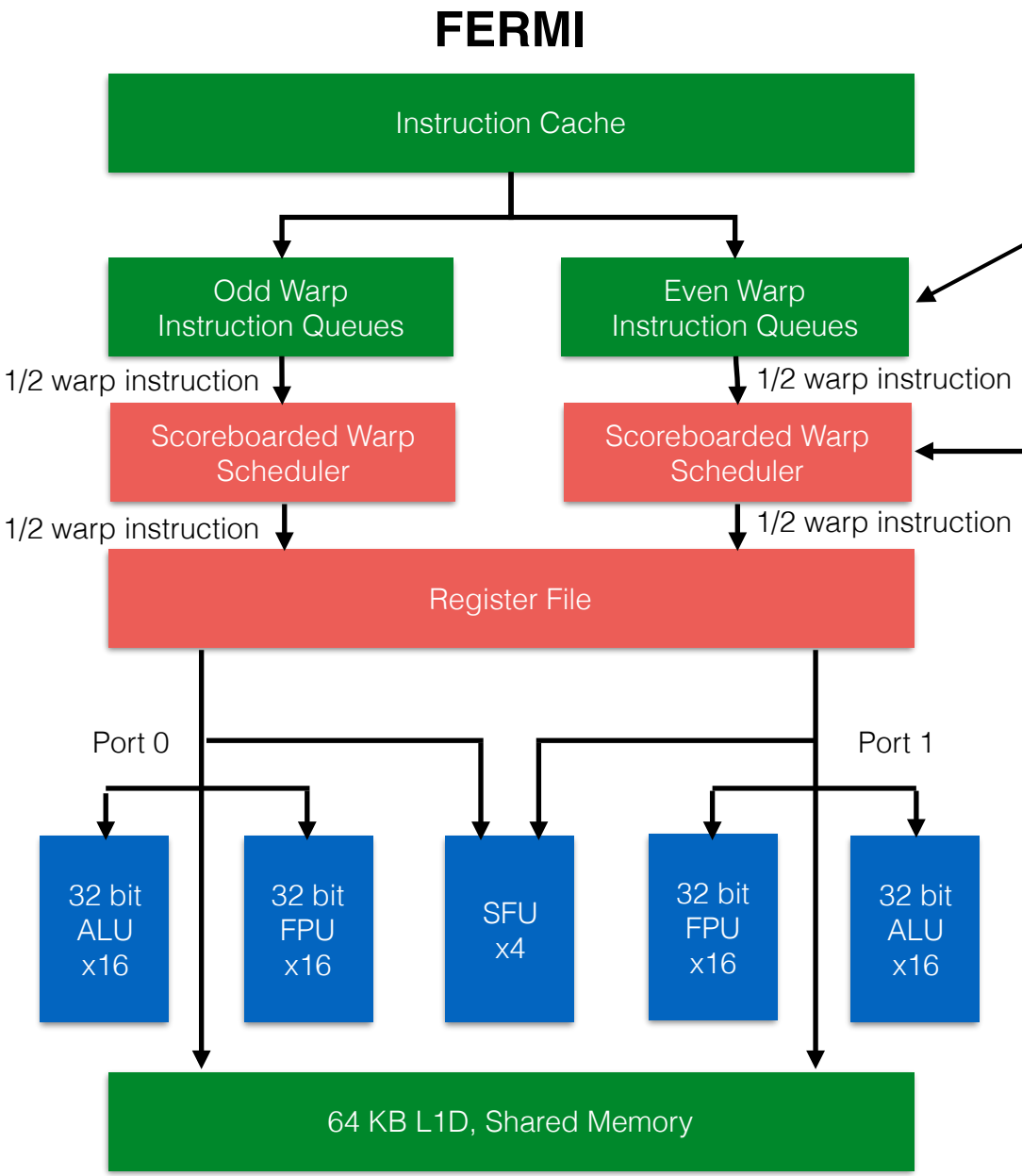
8-32 blocks / SM

CUDA execution (4)



48-64 warps /SM

CUDA execution (5)



48 queues,
1 for each warp

Scoreboard:
Track availability of operands handle structural hazards (FP64, SFU, …)

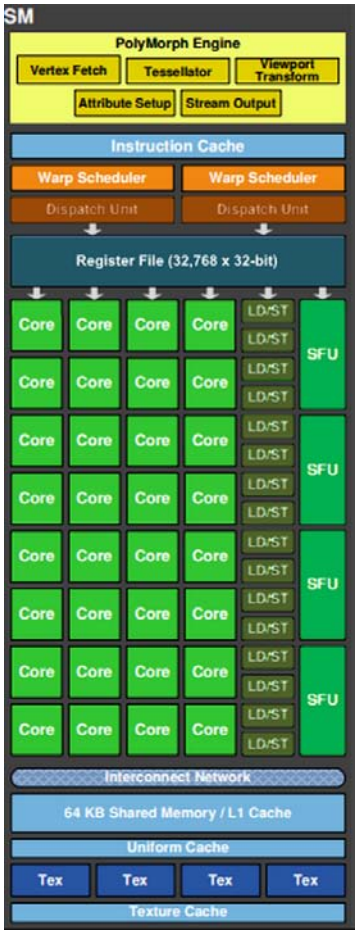
Scheduler complexity:
Large variety of instructions with different execution latency (factor 4)

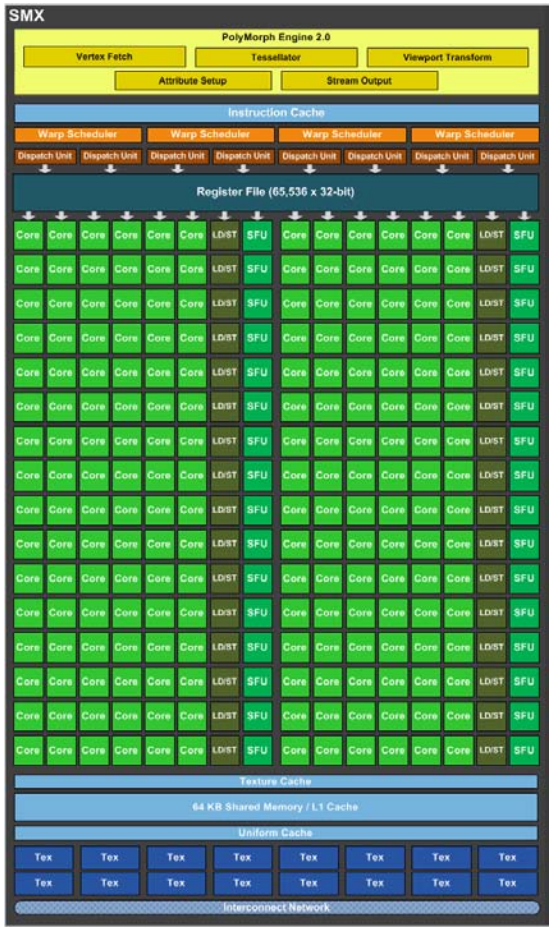
Handle priority (ex: register usage), ready to execute warp, mark certain queues as ‘not ready’ based on expected latency of instruction

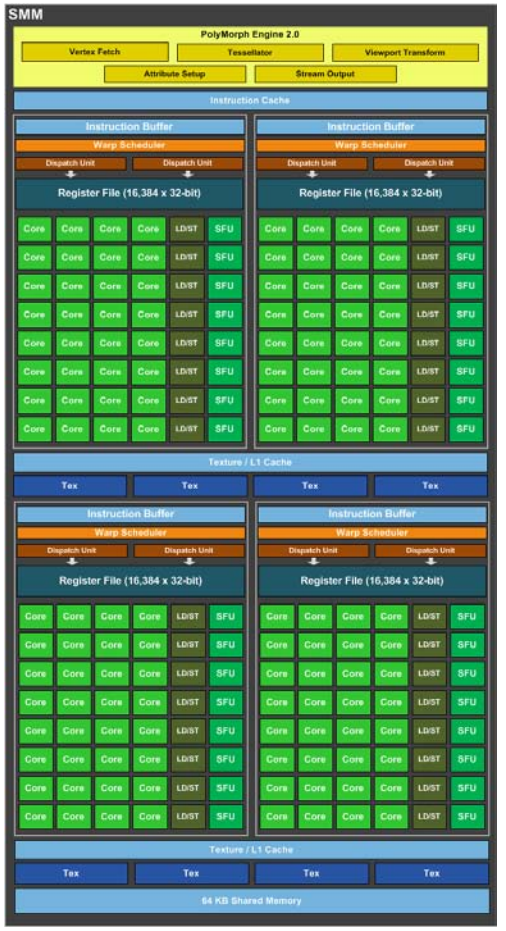
Subtleties (6)

- Operand collector / Temporal register cache
  - Hardware which read in many register values and buffer them for later consumption.
  - Could be useful for broadcasting values to FU
- Result queue
  - Similar concept but for results
  - Could be used for forwarding results to FU (CPU)

Evolution: Fermi, Kepler, Maxwell







Simpler scoreboard

- No register dependency analysis
- Only keep track of long latency inst.

Some figures

Maximum number of threads per block	1024
Maximum number of resident blocks per multiprocessor	8-32
Maximum number of resident warps per multiprocessor	48-64
Number of 32-bit registers per multiprocessor	32K-128K

Topics covered

- Definition of the scheduling problem
- Scheduling on superscalar architecture
- Scheduling on vector architecture
- **How to schedule**
- New problems

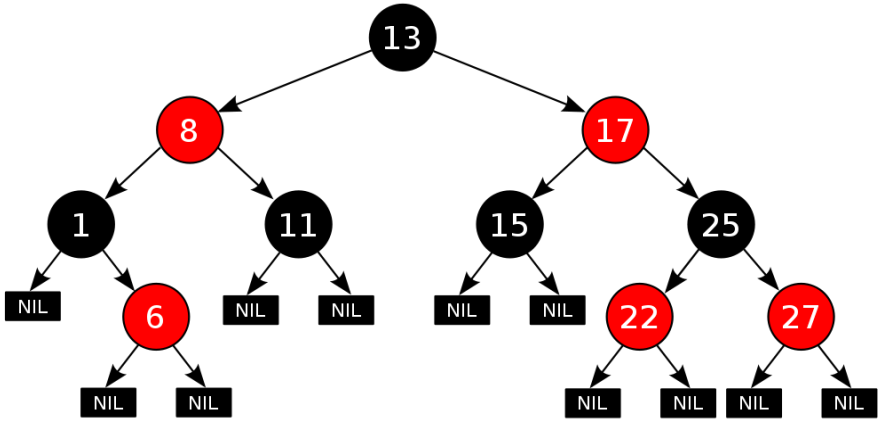
Scheduling in GPU's

- Stream are scheduled among GPUs
  ex: **Red/Black tree** (Used at OS Level)
- Kernel of a Stream are scheduler on a given GPUs using **FIFO**
- Block are scheduled among SMs with a **round robin** algorithm
- Warp are scheduled to SIMT processor using a **round robin** algorithm with **priority** and **aging**
- Data dependencies are handle using a **scoreboard**

Scheduling at OS level

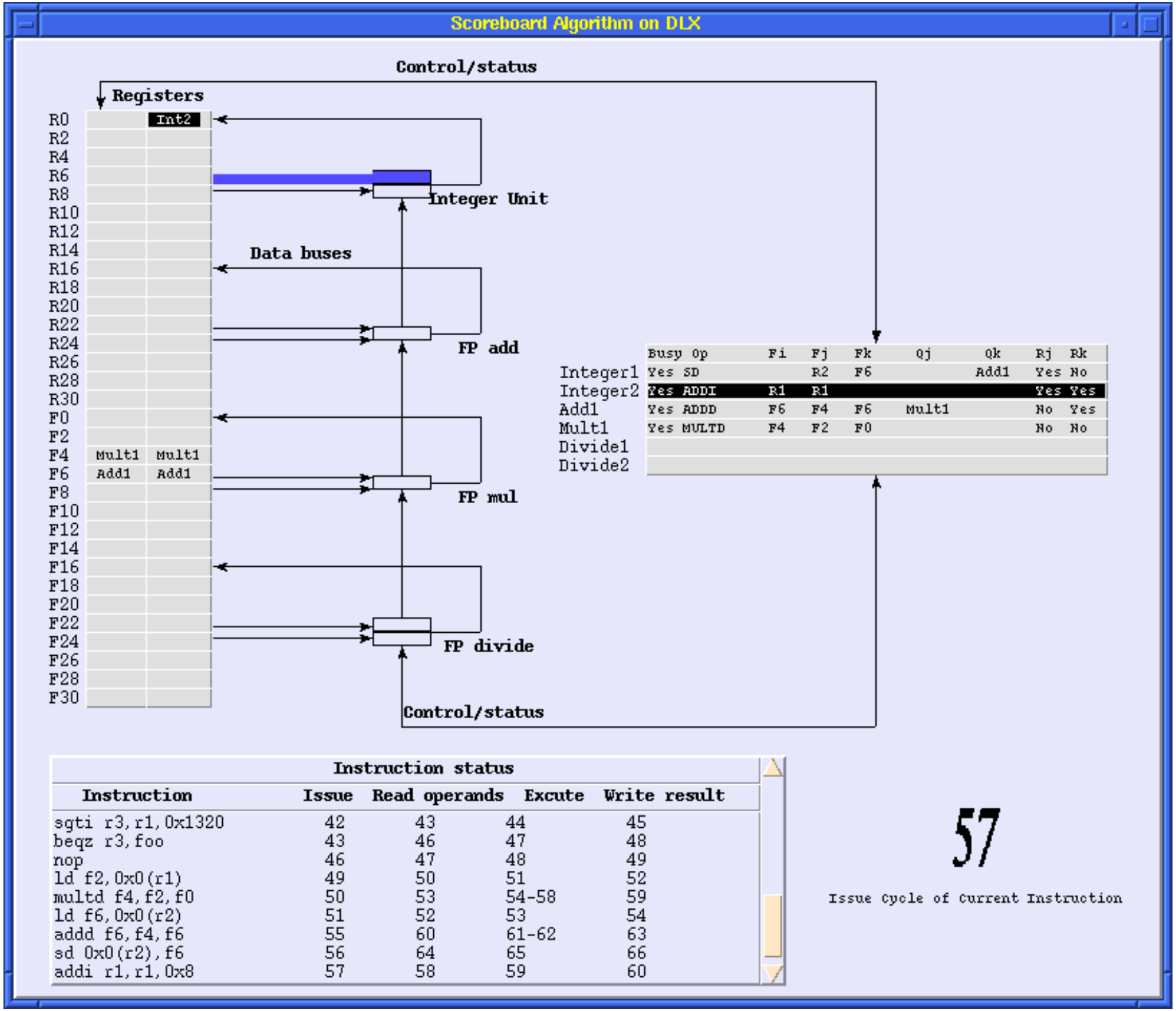
- Example: Completely Fair Scheduler (CFS, Linux kernel 2.6.23)
  - Maximize CPU usage & performance
  - Based on Red/Black tree indexed with execution time

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)



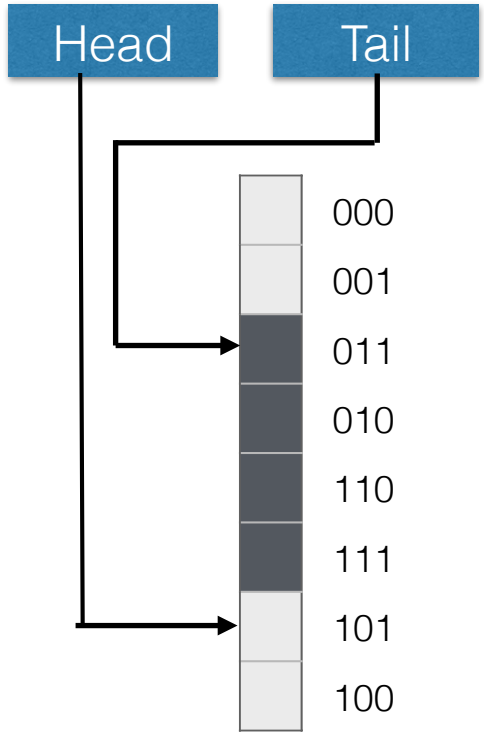
Scoreboard

- Keeps track of dependencies, state or operations
- Instruction status (which steps the instruction is in)
  - Issue, Read Operands, Execution, Write Results
- Function unit status (For each functional unit)
  - Busy: Indicates whether the unit is busy or not
  - Op: Operation to perform in the unit (e.g., + or –)
  - Fi: Destination register
  - Fj, Fk: Source-register numbers
  - Qj, Qk: Functional units producing source registers Fj, Fk
  - Rj, Rk: Flags indicating when Fj, Fk are ready
- Register result status—Indicates which functional unit will write each register,
  - If one exists. Blank when no pending instructions will write that register


Scoreboard Algorithm on DLX
Control/status
Registers
R0
R2
R4
R6
R8
R10
R12
R14
R16
R18
R20
R22
R24
R26
R28
R30
F0
F2
F4
F6
F8
F10
F12
F14
F16
F18
F20
F22
F24
F26
F28
F30
Int2
Mult1
Mult1
Add1
Add1
Integer Unit
Data buses
FP add
FP mul
FP divide
Control/status
Busy Op Fi Fj Fk Qj Qk Rj Rk
Integer1 Yes SD R2 F6 Add1 Yes No
Integer2 Yes ADDI R1 R1 Yes Yes
Add1 Yes ADDD F6 F4 F6 Mult1 No Yes
Mult1 Yes MULTD F4 F2 F0 No No
Divide1
Divide2
Instruction status
Instruction Issue Read operands Excute Write result
sgti r3,r1,0x1320 42 43 44 45
beqz r3,foo 43 46 47 48
nop 46 47 48 49
ld f2,0x0(r1) 49 50 51 52
multd f4,f2,f0 50 53 54-58 59
ld f6,0x0(r2) 51 52 53 54
addd f6,f4,f6 55 60 61-62 63
sd 0x0(r2),f6 56 64 65 66
addi r1,r1,0x8 57 58 59 60
57
Issue Cycle of Current Instruction

FIFO

- First implementation in electronics of FIFO: 1969 Fairchild Semiconductor by Peter Alfke.
- Queues processes in the order that they arrive in the queue
  - Minimal overhead
  - No prioritization
- Implementations
  - Hardware Shift Register
  - Memory Structure
  - Circular buffer
  - List
- Hardware
  - Set of Read and Write pointers, Storage (dual-port SRAM, flip-flops, latches), Control logic
  - Synchronous (same clock)
  - Asynchronous (stability pb, use of gray code for read/write pointers to ensure correct flag)
  - Flags : full/empty, almost full, almost empty, …

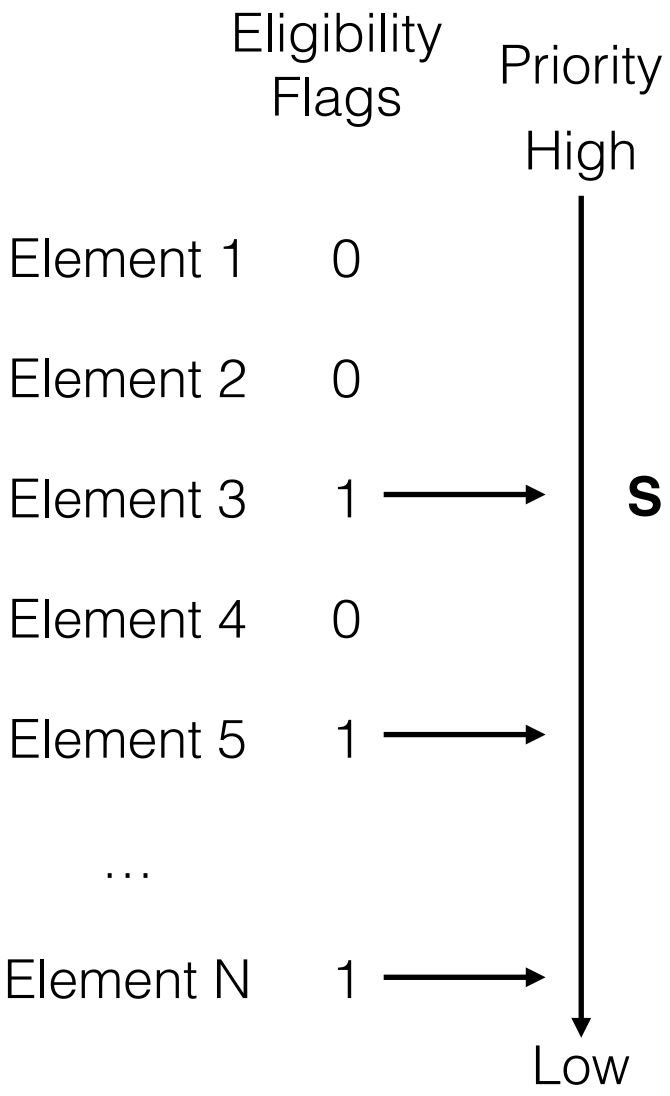


FIFO (2)

- FIFO Empty Flag:
  - When the read address register reaches the write address register, the FIFO triggers the Empty signal.
- FIFO Full Flag:
  - When the write address register reaches the read address register, the FIFO triggers the FULL signal.
- Both case Read @ == Write @
  - Distinguish both cases: add 1 bit to both @ inverted each time the @ wraps
  - R@ = W@ => FIFO empty
  - LSB(R@) = LSB(W@) => FIFO full

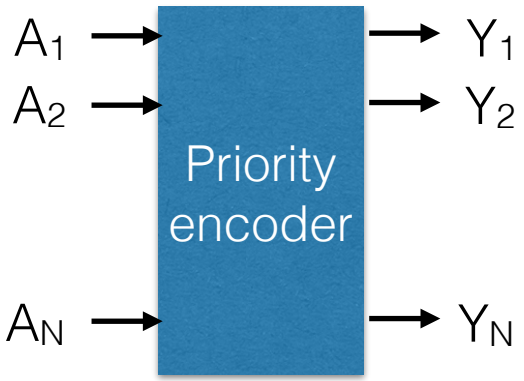
Priority

- Idea:
  Schedule the highest-priority eligible element first
- Implementation: (Static version)
  - Rely on a priority enforcer/ encoder chain of elements with a ripple signal « Nobody above is eligible ». Ripple signal can be replaced with a tree of OR gates to detects the presence of eligible entries (carry lookahead)

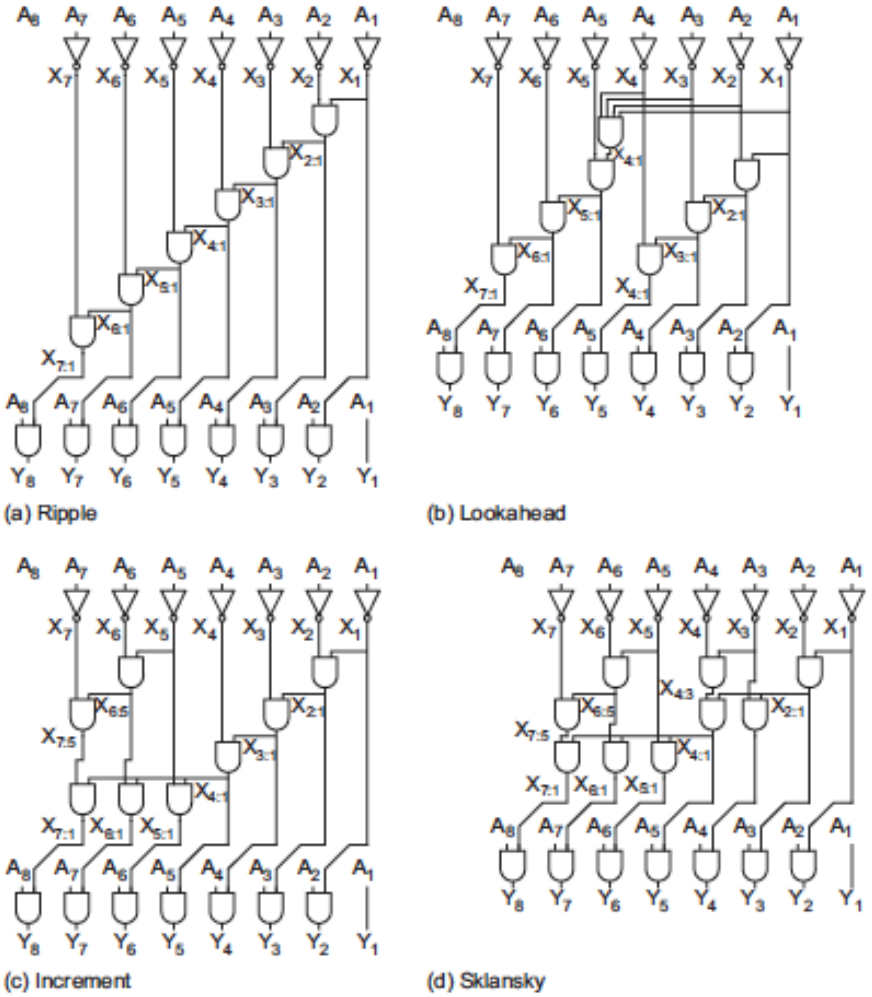


Priority encoder

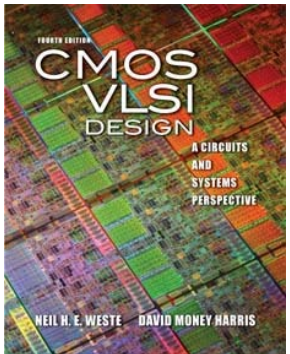
- Each unit i send a request by setting A_i, and receive a granting bit Y_i
  - $Y_1 = A_1$,
  - $Y_2 = A_2 . \overline{A_1}$,
  - $Y_3 = A_3 . \overline{A_2} . \overline{A_1}$,
  - …
  - $Y_N = A_N . \overline{A_{N-1}} . \ldots . \overline{A_2} . \overline{A_1}$,



Priority encoder

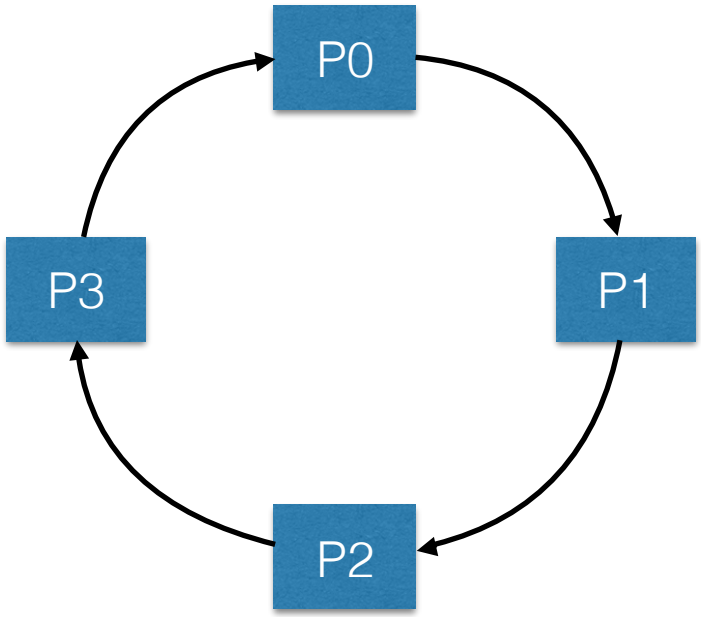


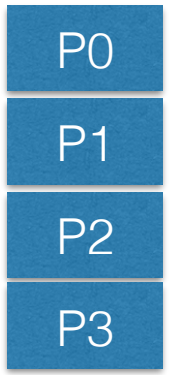
Tradeoffs between delay and gate count



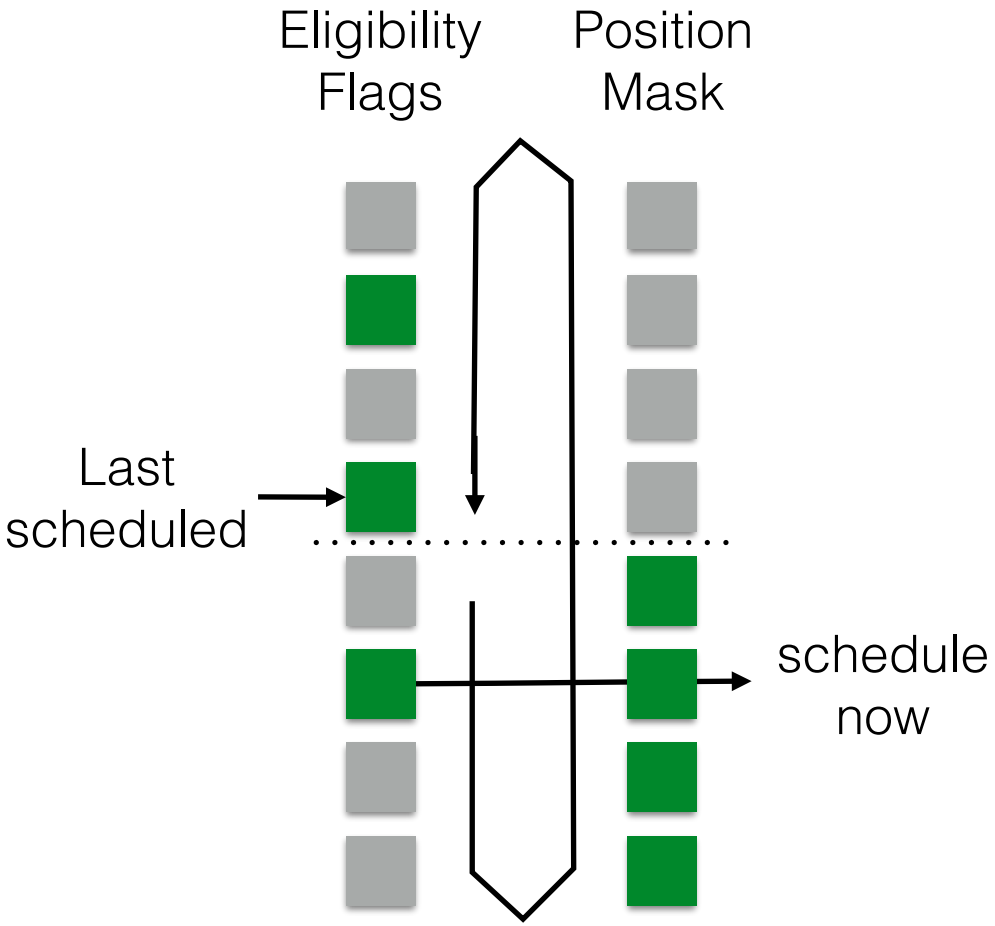
Round Robin

- Time slices are assigned to each process in equal portions and in circular order.
  - No priority
  - Simple, easy to implement, starvation free
  - Known worst case wait time = function of the number of element -1





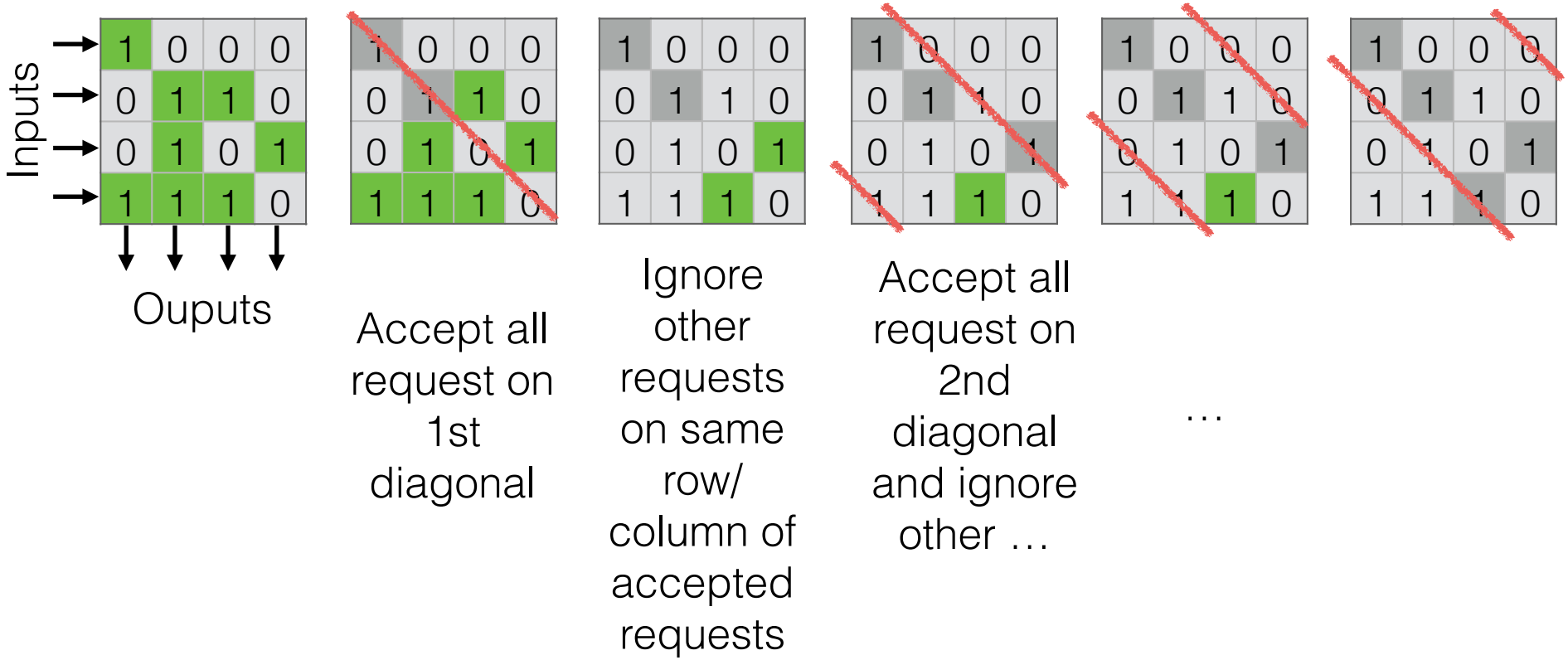
Round-Robin (1)



- 1 element served each cycle
- Circular encoder/enforcer
  - Each element has the capability to break the chain and become « head » element (similar to carry chain)

Round Robin (2)

- Round Robin Matching algorithm
  2-dimensional round-robin scheduler (2DDR)



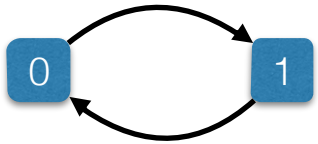
Round-Robin with Fairness

- The first diagonal should be selected in a round robin fashion among {0,1,…N-1}
- Order of applying next diagonals should not always be « left-to-rigth »
  - next_diagonal = (rev_diagonal + offset) mod M
  - offset = ID of first_diagonal +1
  - M = smallest prime number >= N+1

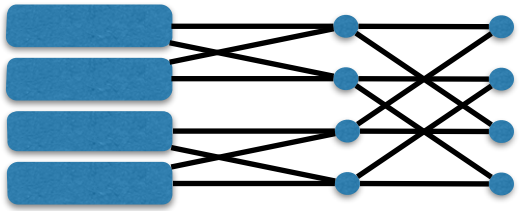
LaMaire, DN Serpanos - IEEE/ACM Transactions on Networking (TON), 1994

Aging

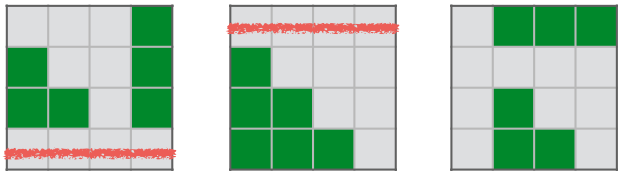
- Implementation N°0: (2 elements)
  - 1 bit counter (alternate between 0 and 1)



- Implementation #1:
  - 64 bit counter, incremented after every instruction
  - Select the element with the oldest counter and reset it. rely on sorting networks

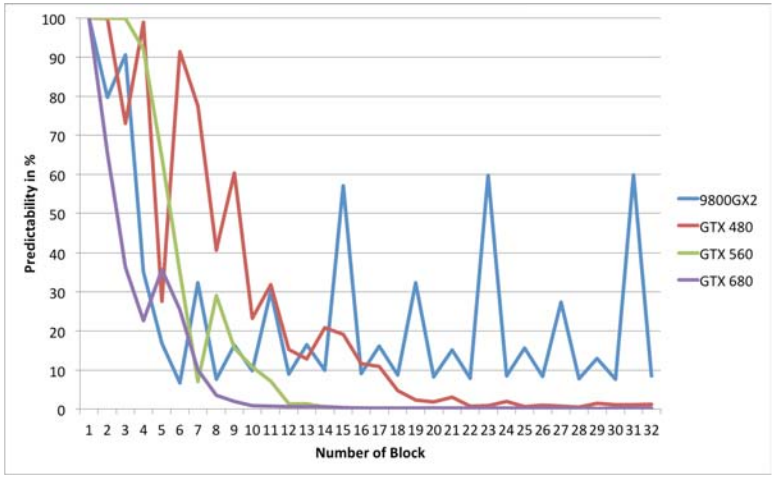


- Implementation #2:
  - Use a matrix M of n x n element initialized at 0
  - Oldest : Select row k with lowest number of 1
    - Set all bits in row k of M to 1
    - Set all bits in column k of M to 0



New problems

- Loss of determinism
  - Complicates debugging, test of correctness
- Key question
  - How to avoid structural hazard and those problems at reasonable cost



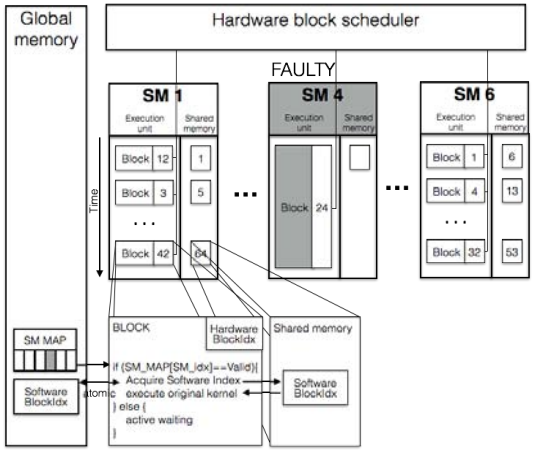
Determinism

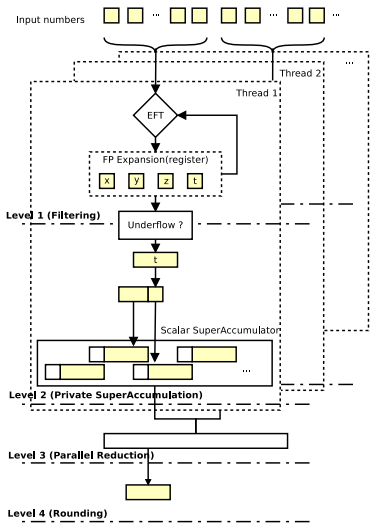
- Various solutions
  - Langage (SHIM, NESL, HASKELL, …)
  - Compiler
  - Software /Algorithm
  - Hardware (Grace, Dthreads)

Example: Algorithmic level

- Numerical issue with FP numbers
  - Non-associativity of FP Numbers
    (0.1 + 100) - 100 != 0.1 + (100-100)

- Numerical reproducibility
  - Weak
    - Enforce an execution order
    - Ex: GPUBurn
  - Strong
    - Independent of the arch., ~correctly rounded
    - Ex: Hierarchical Kulisch Accumulator





Conclusions

- GPU's aren't simple processors
  - Scheduling is as challenging as in traditional CPU
  - Hardware solutions mostly unknown (only guess)
- Trend
  - Toward simpler solution
    (ex: simpler scoreboard in Kepler)
- There is a need for deterministic execution

Links

- Hardware description
  - Rise of the Graphics Processor: http://ieeexplore.ieee.org/ieee_pilot/articles/96jproc05/96jproc05-blythe/article.html
  - http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
  - http://www.realworldtech.com/fermi/
  - http://www.clubic.com/article-332014-3-nvidia-geforce-gtx-480-directx-11-nvidia.html
  - http://www.anandtech.com/
  - http://www.beyond3d.com/ …
- Scheduling
  - Network architecture (Manolis Katevenis): http://www.csd.uoc.gr/~hy534/
  - « An Investigation of the performance of various dynamic scheduling techniques »: http://hps.ece.utexas.edu/pub/butler_micro25.pdf
- Books