Multi-threading or SIMD?

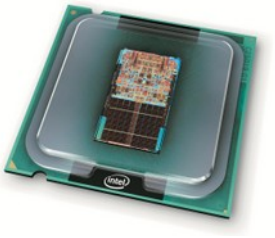
How GPU architectures exploit regularity

Caroline Collange
Arénaire, LIP, ENS de Lyon
caroline.collange@inria.fr

ARCHI'11
June 14, 2011

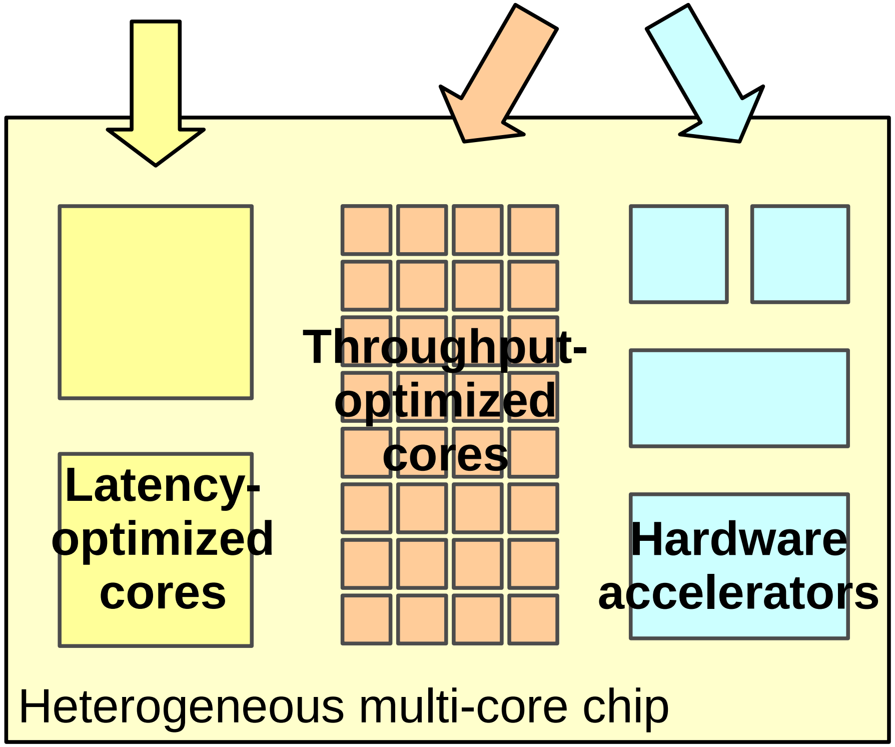
From GPU to integrated many-core

- Yesterday (2000-2010)
  - Homogeneous multi-core
  - Discrete components
- Today (2011-...)
  Heterogeneous multi-core
  - Intel Sandy Bridge
  - AMD Fusion
  - NVIDIA Denver/Maxwell project…
- Focus on the throughput-optimized part
  - Similarities?
  - Differences?
  - Possible improvements?



Central Processing Unit (CPU)

Graphics Processing Unit (GPU)



Latency-optimized cores

Throughput-optimized cores

Hardware accelerators

Heterogeneous multi-core chip

Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Dealing with variable latency

The 1980': pipelined processor

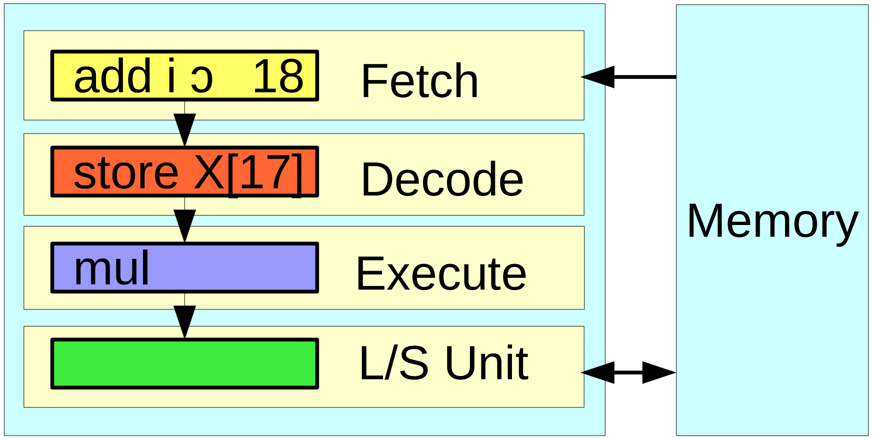
- Example: scalar-vector multiplication: X ɔ a⁄X

```
for i = 0 to n-1
   X[i] ← a * X[i]
```

Source code

```
   move   i ← 0
loop:
   load   t ← X[i]
   mul    t ← a×t
   store X[i] ← t
   add    i ← i+1
   branch i<n? loop
```

Machine code



Sequential CPU

The 1990': superscalar processor

- Goal: improve performance of sequential applications
  - Latency: time to get the result
- Exploits Instruction-Level Parallelism (ILP)
- Lots of tricks
  - Branch prediction, out-of-order execution, register renaming, data prefetching, memory disambiguation…
- Basis: speculation
  - Take a bet on future events
  - If right: time gain
  - If wrong, roll back: energy loss

What makes speculation work: regularity

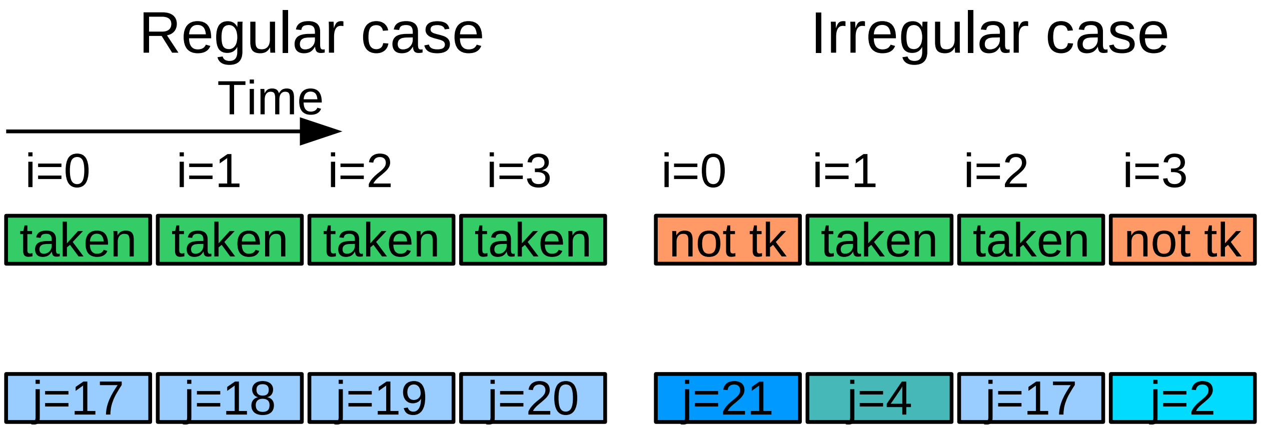
- Application behavior likely to follow regular patterns

Control regularity

Memory regularity

```
for(i…)
{
    if(f(i)) {
    }

    j = g(i);
    x = a[j];
}
```



	Regular case				Irregular case			
	i=0	i=1	i=2	i=3	i=0	i=1	i=2	i=3
Control regularity	taken	taken	taken	taken	not tk	taken	taken	not tk
Memory regularity	j=17	j=18	j=19	j=20	j=21	j=4	j=17	j=2

- Applications
    - Caches
    - Branch prediction
    - Instruction prefetch, data prefetch, write combining…

The 2000': going multi-threaded

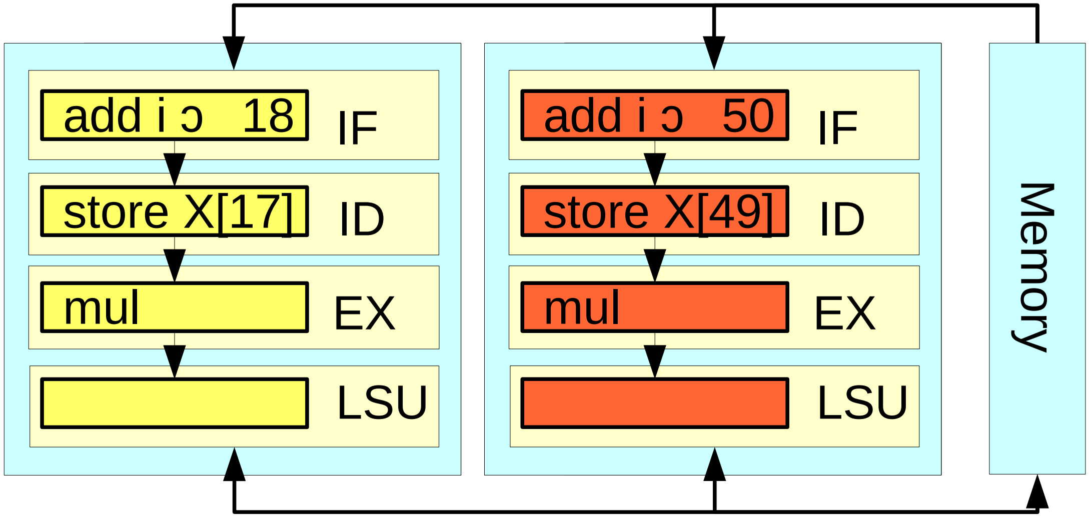
- Obstacles to continuous CPU performance increase
  - Power wall
  - Memory wall
  - ILP wall
- 2000-2010: gradual transition from latency-oriented to throughput-oriented
  - Homogeneous multi-core
  - Interleaved multi-threading
  - Clustered multi-threading

Homogeneous multi-core

- Replication of the complete execution engine
- Multi-threaded software

```
    move  i ← slice_begin
loop:
    load  t ← X[i]
    mul   t ← a×t
    store X[i] ← t
    add   i ← i+1
    branch i<slice_end? loop
```

Machine code



Threads: T0 T1

- Improves throughput thanks to explicit parallelism

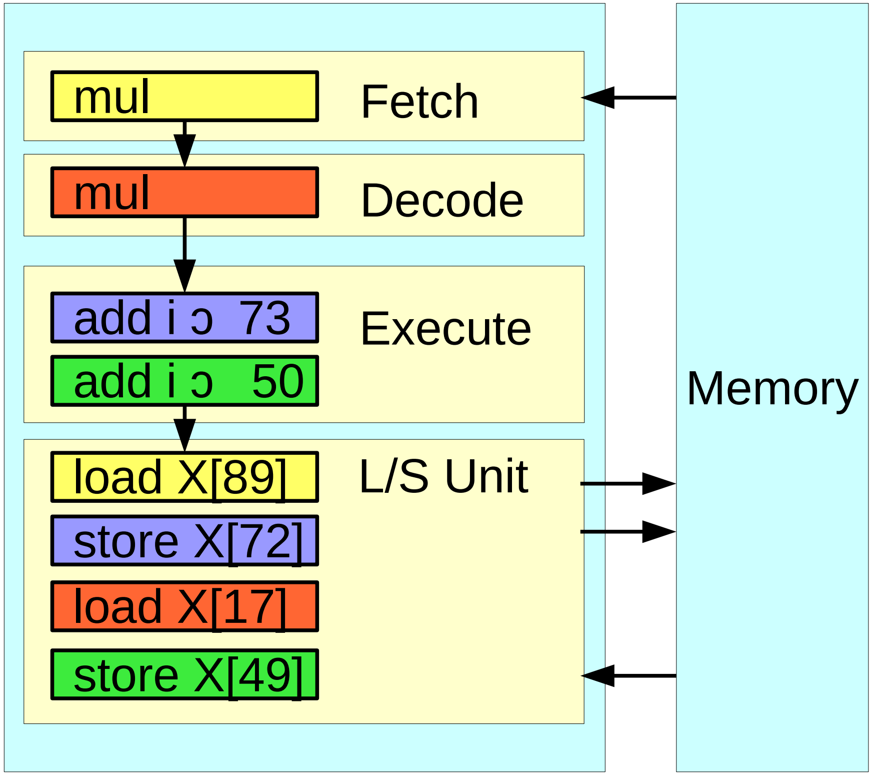
Interleaved multi-threading

- Time-multiplexing of processing units
- Same software view

```
    move  i ← slice_begin
loop:
    load  t ← X[i]
    mul   t ← a×t
    store X[i] ← t
    add   i ← i+1
    branch i<slice_end? loop
```

Machine code

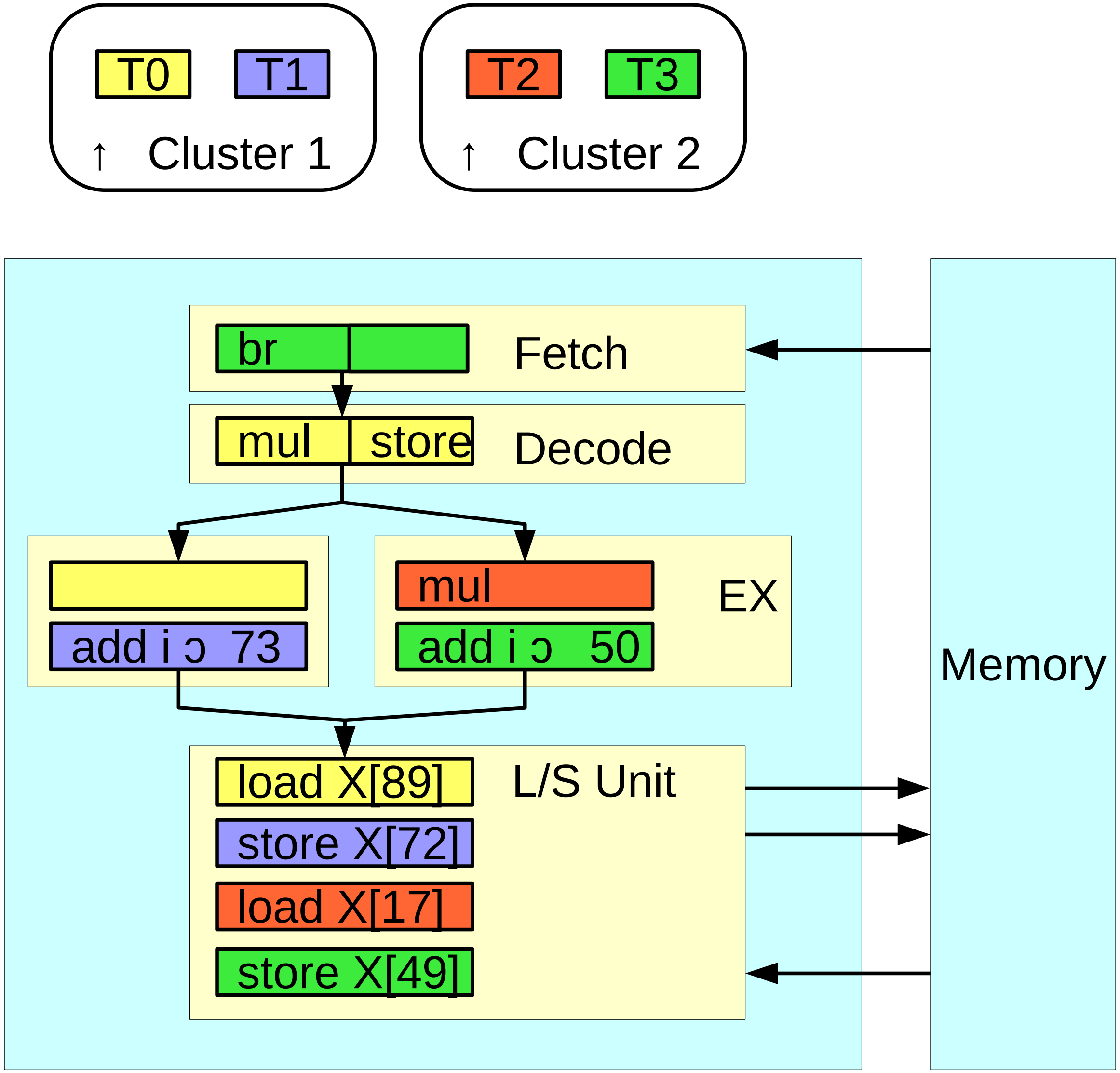
Threads: T0 T1 T2 T3



- Hides latency thanks to explicit parallelism

Clustered multi-core

- For each individual unit, select between
  - Replication
  - Time-multiplexing
- Examples
  - Sun UltraSparc T2
  - AMD Bulldozer



- Area-efficient tradeoff

Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Dealing with variable latency

Heterogeneity: causes and consequences

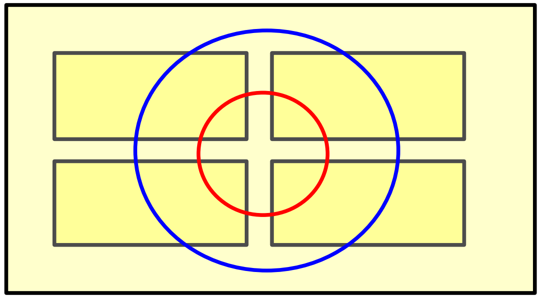
- Amdahl's law

$$S=\frac{1}{(1-P)+\frac{P}{N}}$$

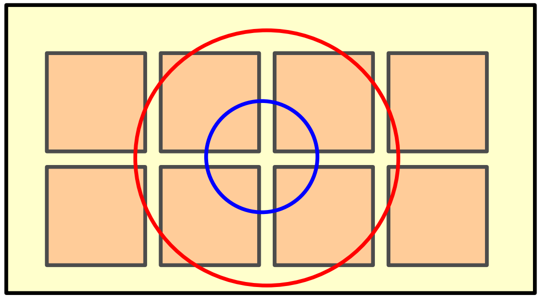
Time to run sequential portions

Time to run parallel portions

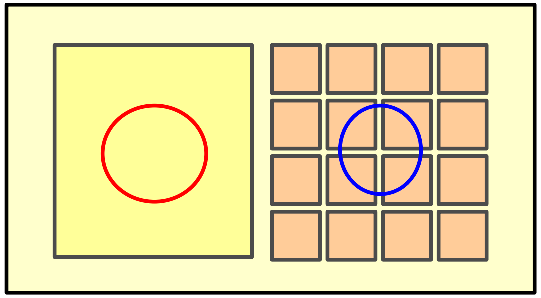
- Latency-optimized multi-core
  - Low efficiency on parallel portions: spends too much resources



- Throughput-optimized multi-core
  - Low performance on sequential portions

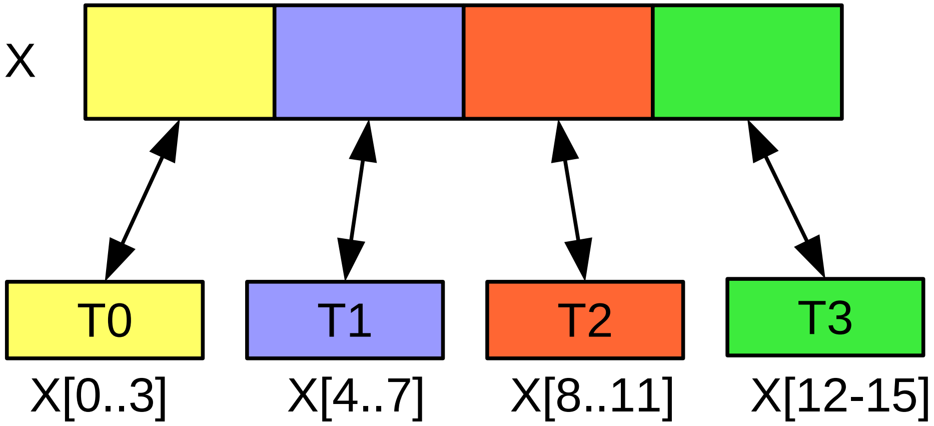


- Heterogeneous multi-core
  - Power-constrained: can afford idle transistors
  - Suggests more radical specialization

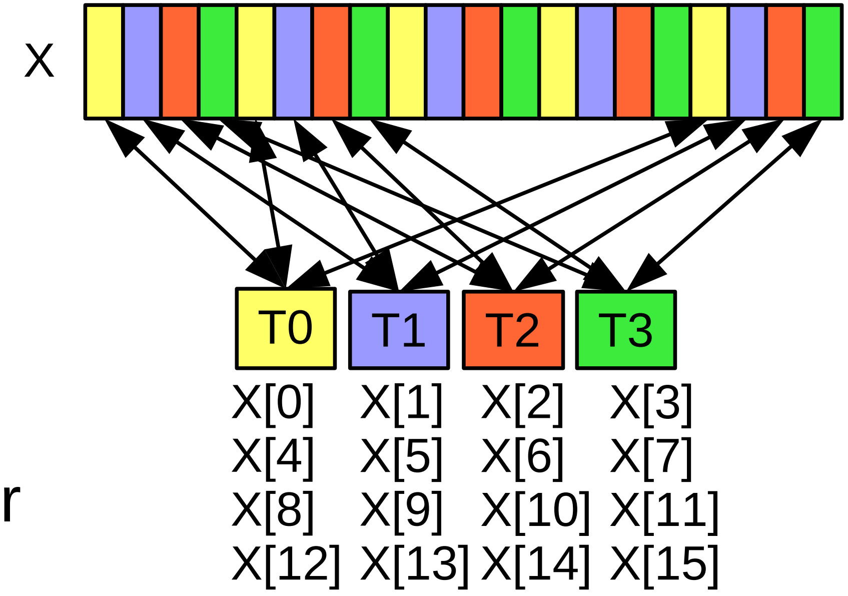


Threading granularity

- Coarse-grained threading
  - **Decouple** tasks to reduce **conflicts** and inter-thread communication

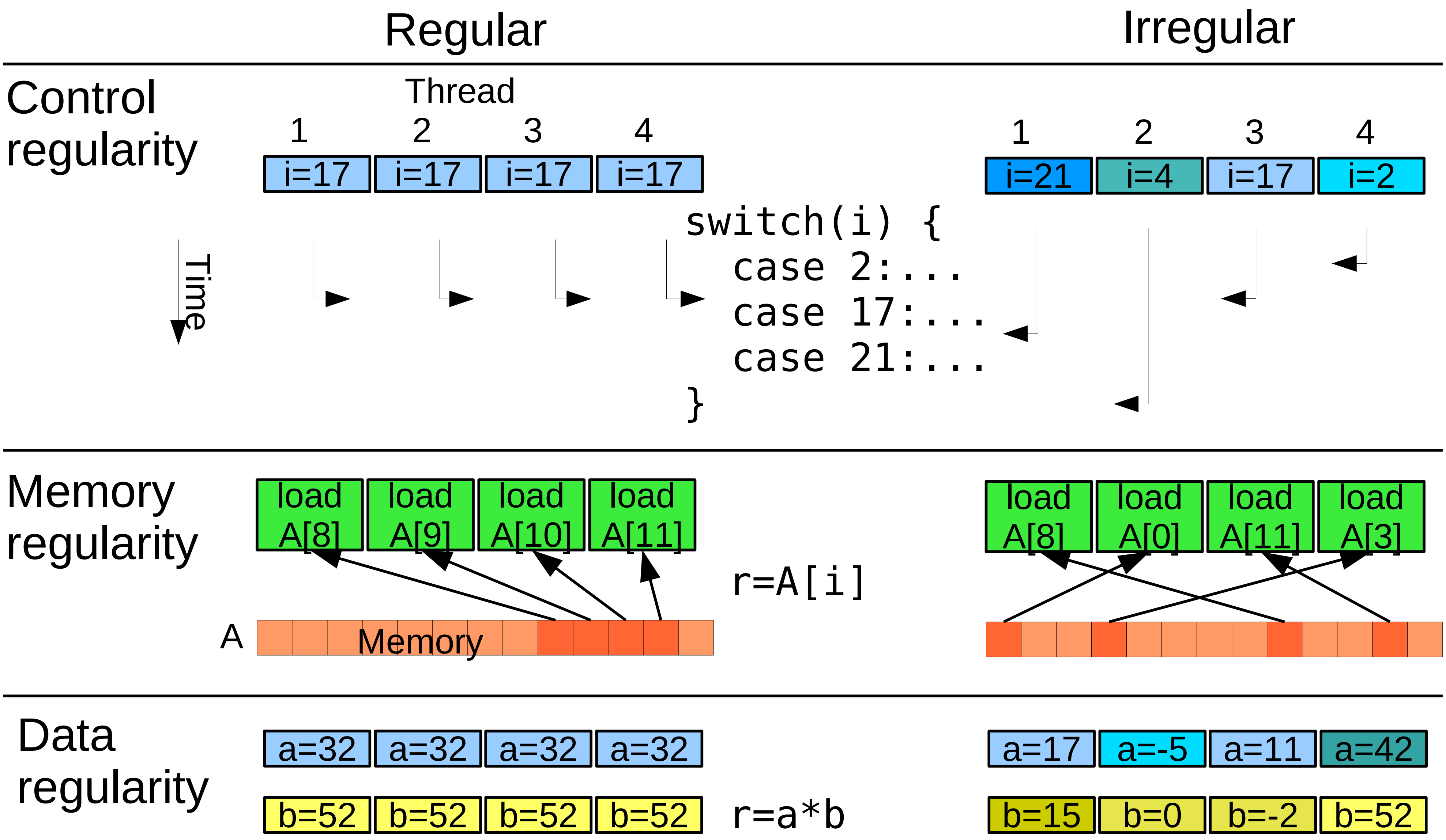


- Fine-grained threading
  - **Interleave** tasks
  - Exhibit **locality**: neighbor threads share memory
  - Exhibit **regularity**: neighbor threads have a similar behavior



Parallel regularity

- Similarity in behavior between threads



Regular

Irregular

Control regularity

```
switch(i) {
  case 2:...
  case 17:...
  case 21:...
}
```

Memory regularity

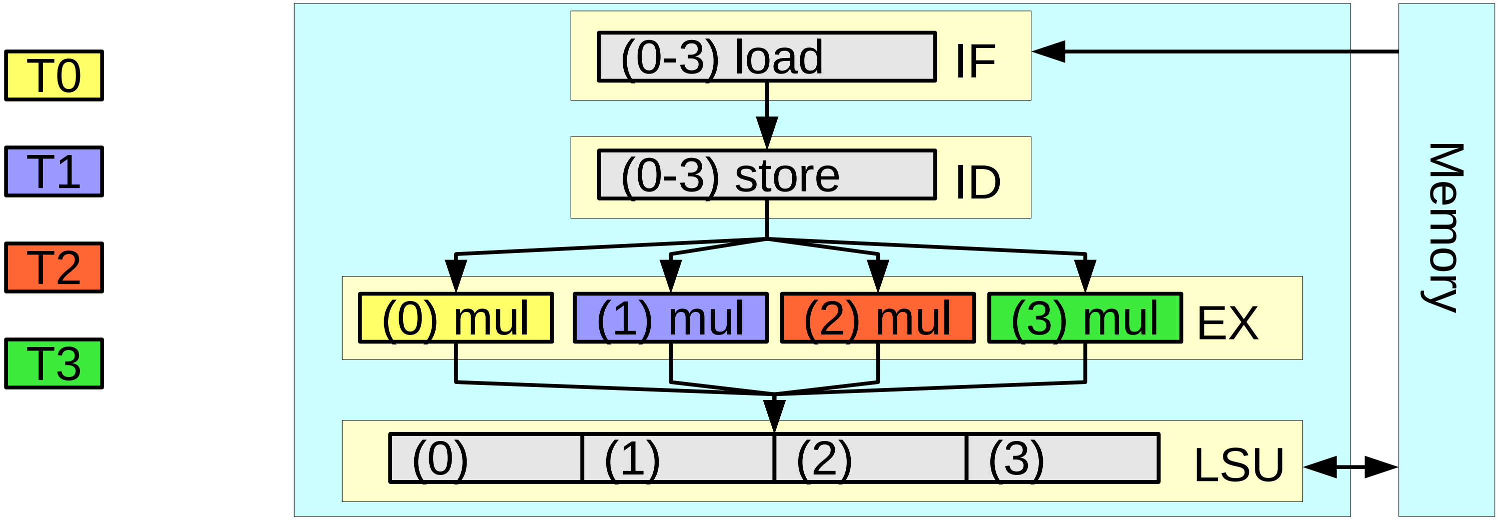
```
r=A[i]
```

Data regularity

```
r=a*b
```

Single Instruction, Multiple Threads (SIMT)

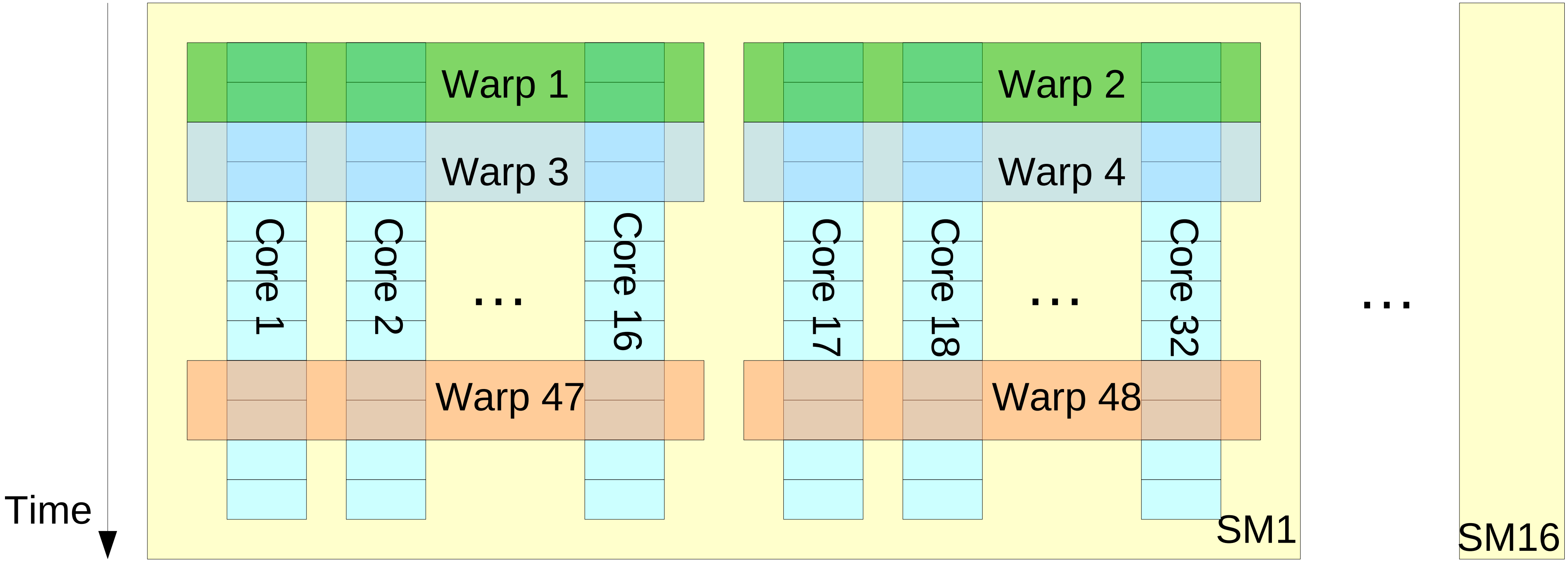
- Cooperative sharing of fetch/decode, load-store units
  - Fetch 1 instruction on behalf of several threads
  - Read 1 memory location and broadcast to several registers



- In NVIDIA-speak
  - SIMT: Single Instruction, Multiple Threads
  - Convoy of synchronized threads: *warp*
- Improves Area/Power-efficiency thanks to regularity
  - Consolidates memory transactions: less memory pressure

Example GPU: NVIDIA GeForce GTX 580

- SIMT: warps of 32 threads
- 16 SMs / chip
- 2×16 cores / SM, 48 warps / SM



- 1580 Gflop/s
- Up to 24576 threads in flight

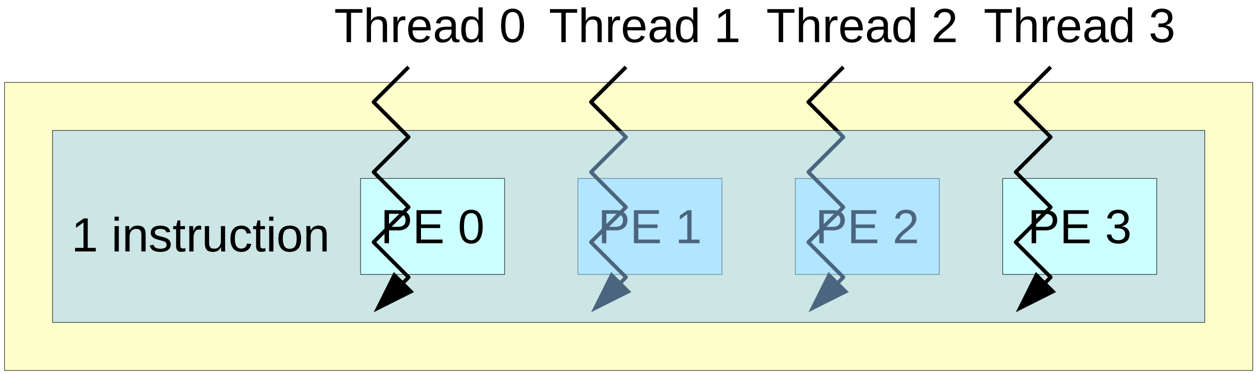
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Dealing with variable latency

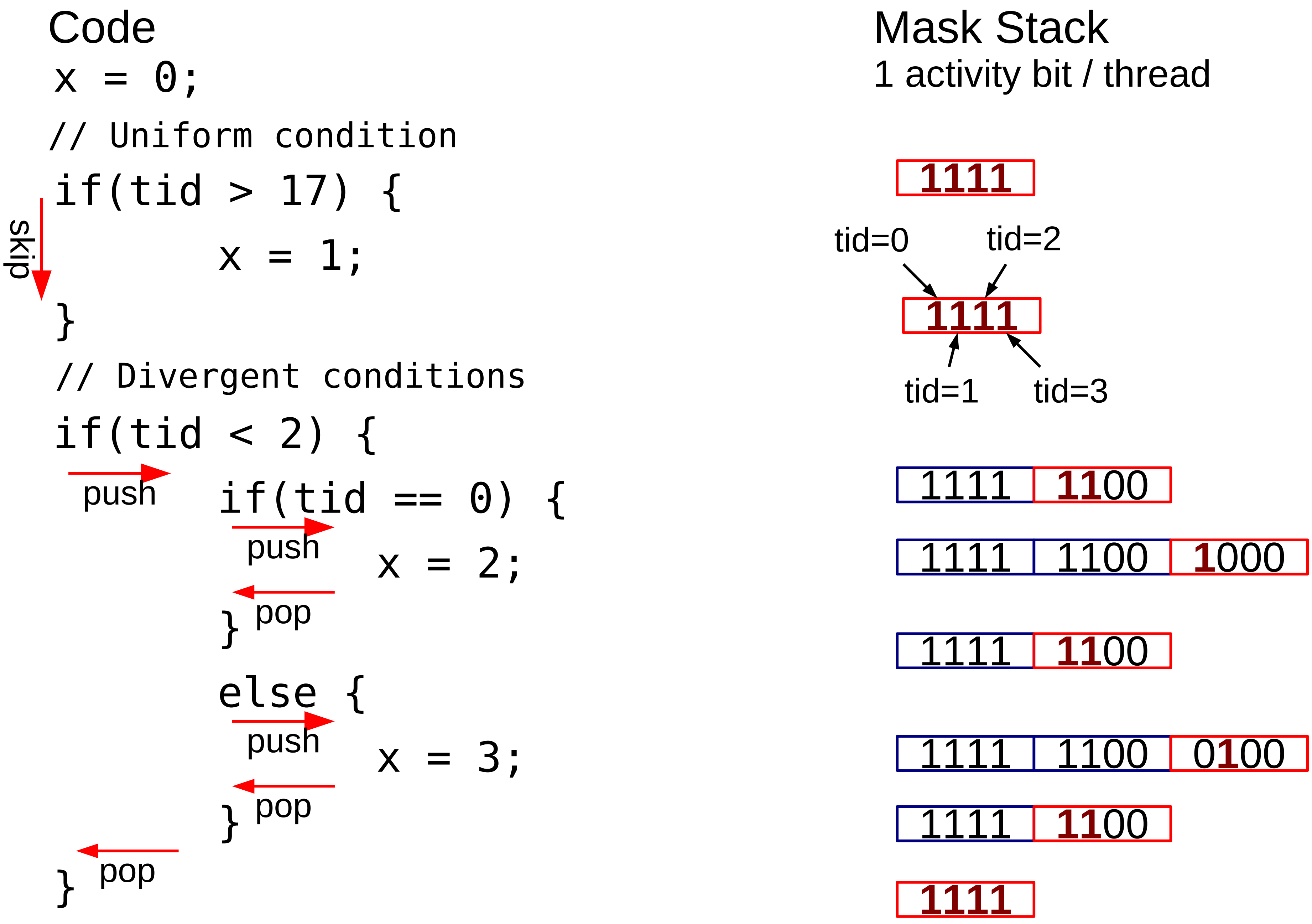
Capturing instruction regularity

- How to handle control divergence?
  - Techniques from Single Instruction, Multiple Data (SIMD) architectures
- Rules of the game
  - One thread per Processing Element (PE)
  - All PE execute the same instruction
  - PEs can be individually disabled

```
x = 0;
// Uniform condition
if(tid > 17) {
        x = 1;
}
// Divergent conditions
if(tid < 2) {
        if(tid == 0) {
                x = 2;
        }
        else {
                x = 3;
        }
}
```



Most common: mask stack



Code

```
x = 0;
// Uniform condition
if(tid > 17) {
        x = 1;
}
// Divergent conditions
if(tid < 2) {
        if(tid == 0) {
                x = 2;
        }
        else {
                x = 3;
        }
}
```

Mask Stack
1 activity bit / thread

1111

1111

1111 1100

1111 1100 1000

1111 1100

1111 1100 0100

1111 1100

1111

A. Levinthal and T. Porter. *Chap - a SIMD graphics processor*. SIGGRAPH’84, 1984.

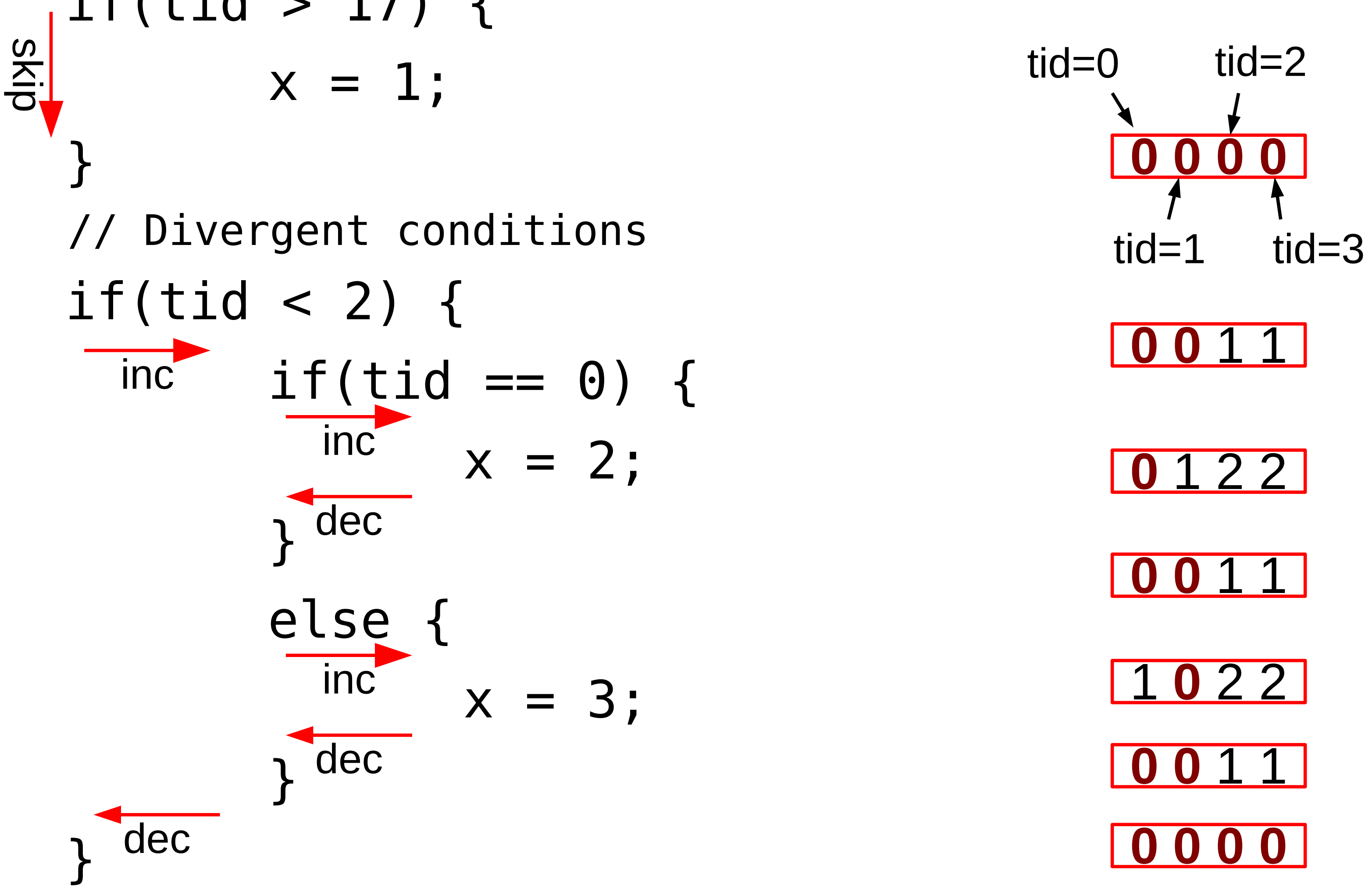
Curiosity: activity counters

Code

```
x = 0;
// Uniform condition
if(tid > 17) {
      x = 1;
}
// Divergent conditions
if(tid < 2) {
      if(tid == 0) {
            x = 2;
      }
      else {
            x = 3;
      }
}
```

Counters

1 (in)activity counter / thread



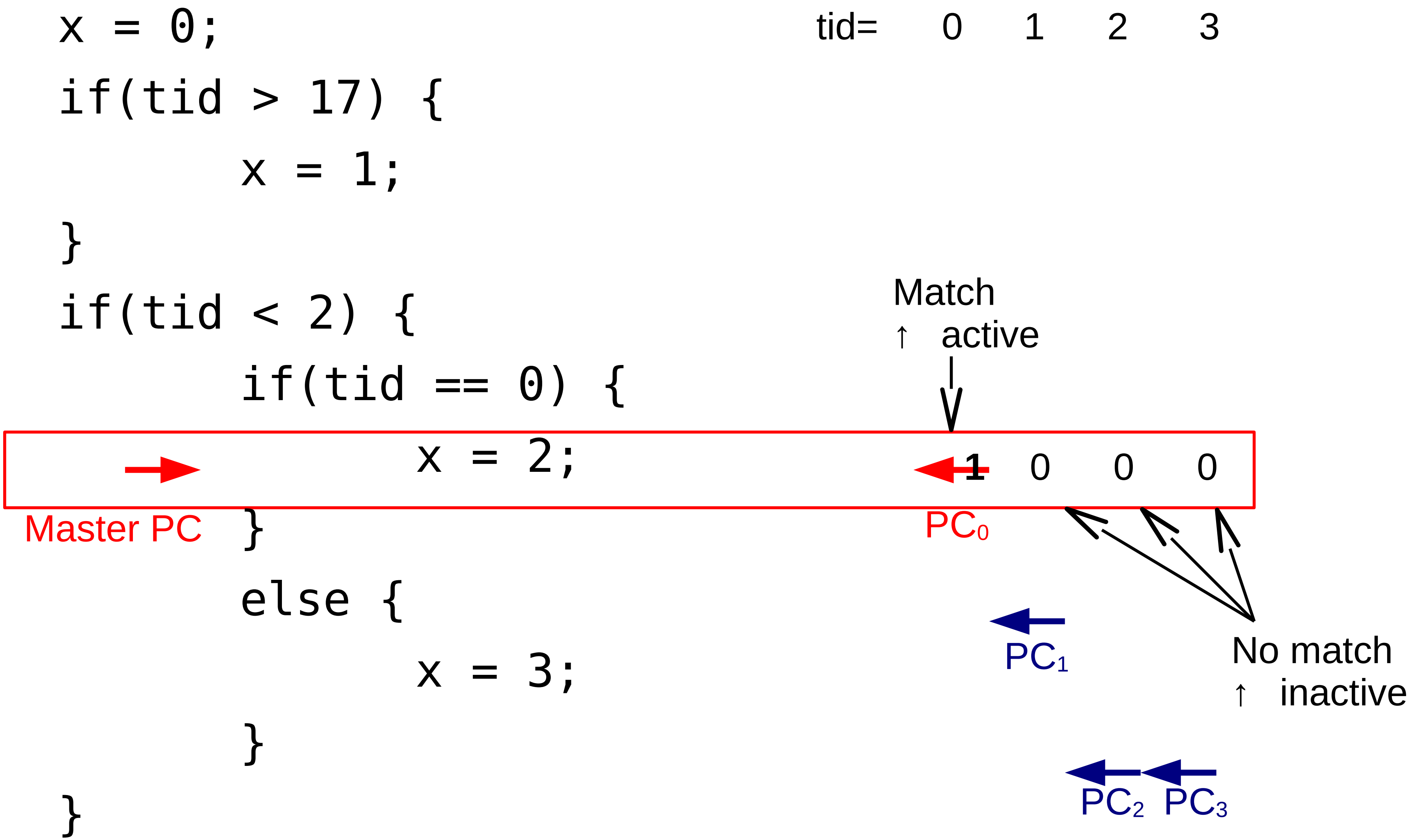
R. Keryell and N. Paris. *Activity counter : New optimization for the dynamic scheduling of SIMD control flow.* ICPP ’93, 1993.

Brute-force: 1 PC / thread

Code

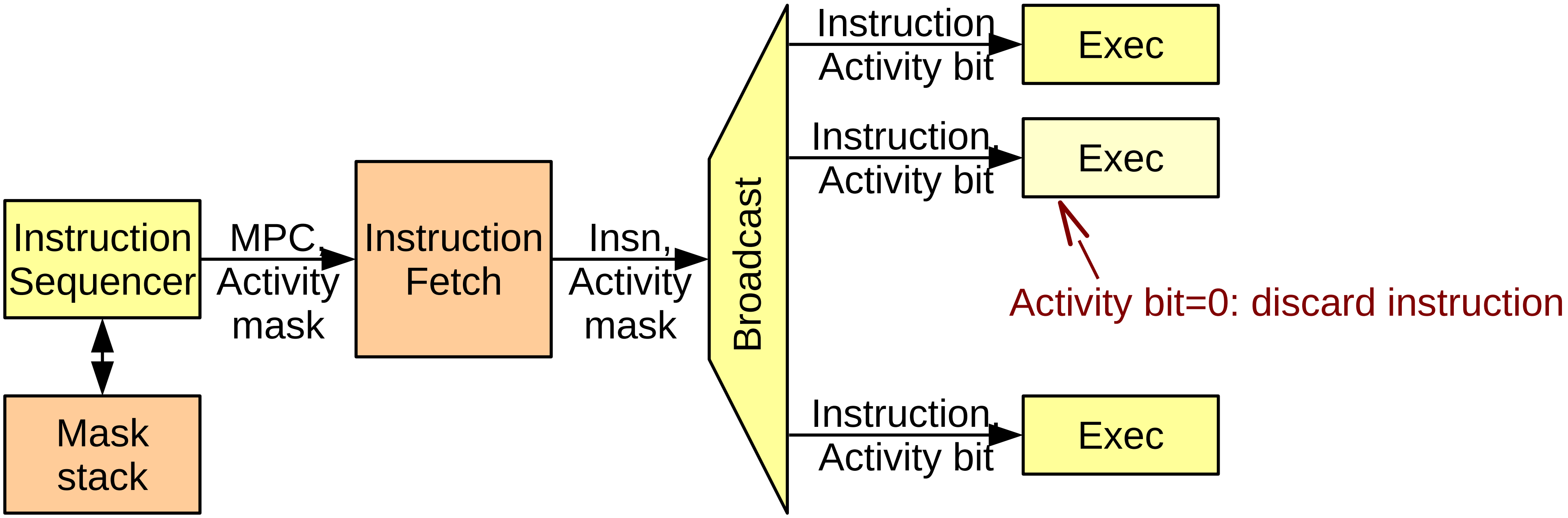
```
x = 0;
if(tid > 17) {
        x = 1;
}
if(tid < 2) {
        if(tid == 0) {
                x = 2;
        }
        else {
                x = 3;
        }
}
```

Program Counters (PCs)



P. Hatcher et al. *A production-quality C* compiler for Hypercube multicomputers,* PPOPP '91, 1991.

Traditional SIMT pipeline


Instruction Sequencer
MPC, Activity mask
Instruction Fetch
Insn, Activity mask
Mask stack
Broadcast
Instruction, Activity bit
Exec
Instruction, Activity bit
Exec
Instruction, Activity bit
Exec
Activity bit=0: discard instruction

- Used in virtually every modern GPU

Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput

Goto considered harmful?

MIPS

```
j
jal
jr
syscall
```

NVIDIA
Tesla
(2007)

```
bar
bra
brk
brkpt
cal
cont
kil
pbk
pret
ret
ssy
trap
.s
```

NVIDIA
Fermi
(2010)

```
bar
bpt
bra
brk
brx
cal
cont
exit
jcal
jmx
kil
pbk
pret
ret
ssy
.s
```

Intel GMA
Gen4
(2006)

```
jmpi
if
iff
else
endif
do
while
break
cont
halt
msave
mrest
push
pop
```

Intel GMA
SB
(2011)

```
jmpi
if
else
endif
case
while
break
cont
halt
call
return
fork
```

AMD
R500
(2005)

```
jump
loop
endloop
rep
endrep
breakloop
breakrep
continue
```

AMD
R600
(2007)

```
push
push_else
pop
loop_start
loop_start_no_al
loop_start_dx10
loop_end
loop_continue
loop_break
jump
else
call
call_fs
return
return_fs
alu
alu_push_before
alu_pop_after
alu_pop2_after
alu_continue
alu_break
alu_else_after
```

AMD Cayman
(2011)

```
push
push_else
pop
push_wqm
pop_wqm
else_wqm
jump_any
reactivate
reactivate_wqm
loop_start
loop_start_no_al
loop_start_dx10
loop_end
loop_continue
loop_break
jump
else
call
call_fs
return
return_fs
alu
alu_push_before
alu_pop_after
alu_pop2_after
alu_continue
alu_break
alu_else_after
```

Control instructions in some CPU and GPU instruction sets

- Why so many?
  - Expose control flow **structure** to the instruction sequencer

SIMD is so last century

- Maspar MP-1 (1990)
  - 1 instruction for 16 384 PEs
  - PE : ~1 mm^2, 1.6 µm process
  - SIMD programming model

/1000

Fewer PEs

×50

Bigger PEs

More divergence

- NVIDIA Fermi (2010)
  - 1 instruction for 16 PEs
  - PE : ~0,03 mm^2, 40 nm process
  - Threaded programming model

→ From centralized control to flexible distributed control

A democratic instruction sequencer

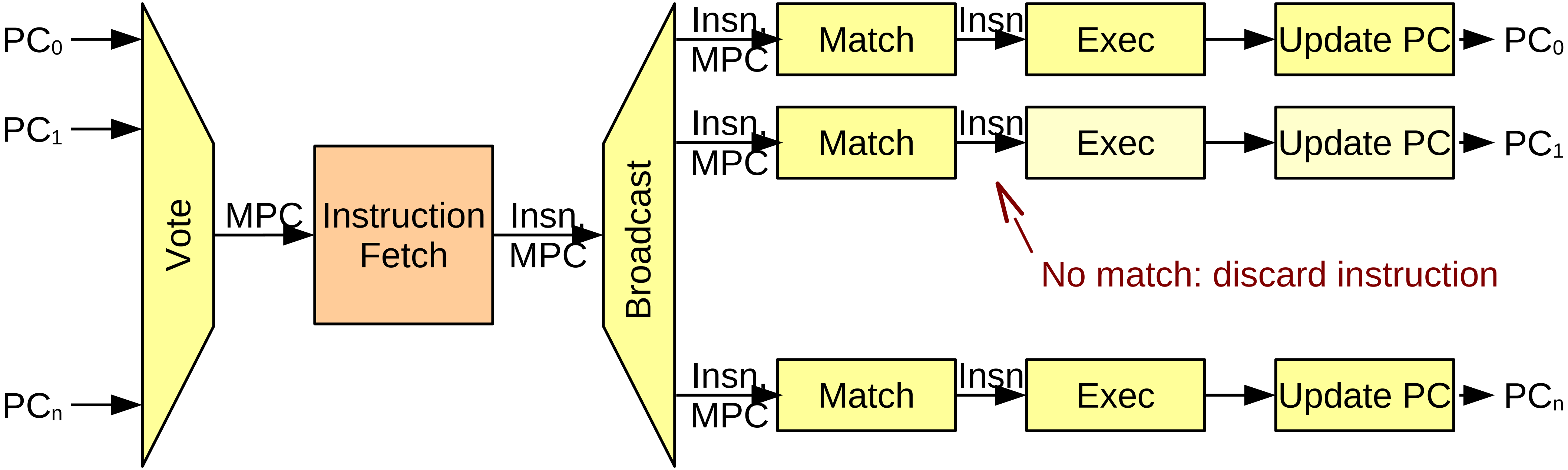
- Maintain one PC per thread
- Vote: select one of the individual PCs as the Master PC
- Which one? Various policies:
    - Majority: most common PC
    - Minimum: threads which are late
    - Deepest control flow nesting level
    - Deepest function call nesting level
    - Various combinations of the former

W. Fung, I. Sham, G. Yuan, and T. Aamodt. *Dynamic warp formation and scheduling for efficient GPU control flow.* MICRO’07, 2007.

J. Meng, D. Tarjan and K. Skadron. *Dynamic warp subdivision for integrated branch and memory divergence tolerance.* ISCA'2010, 2010.

C. Collange. *Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT.* SympA'14, 2011.

Our new SIMT pipeline



No match: discard instruction

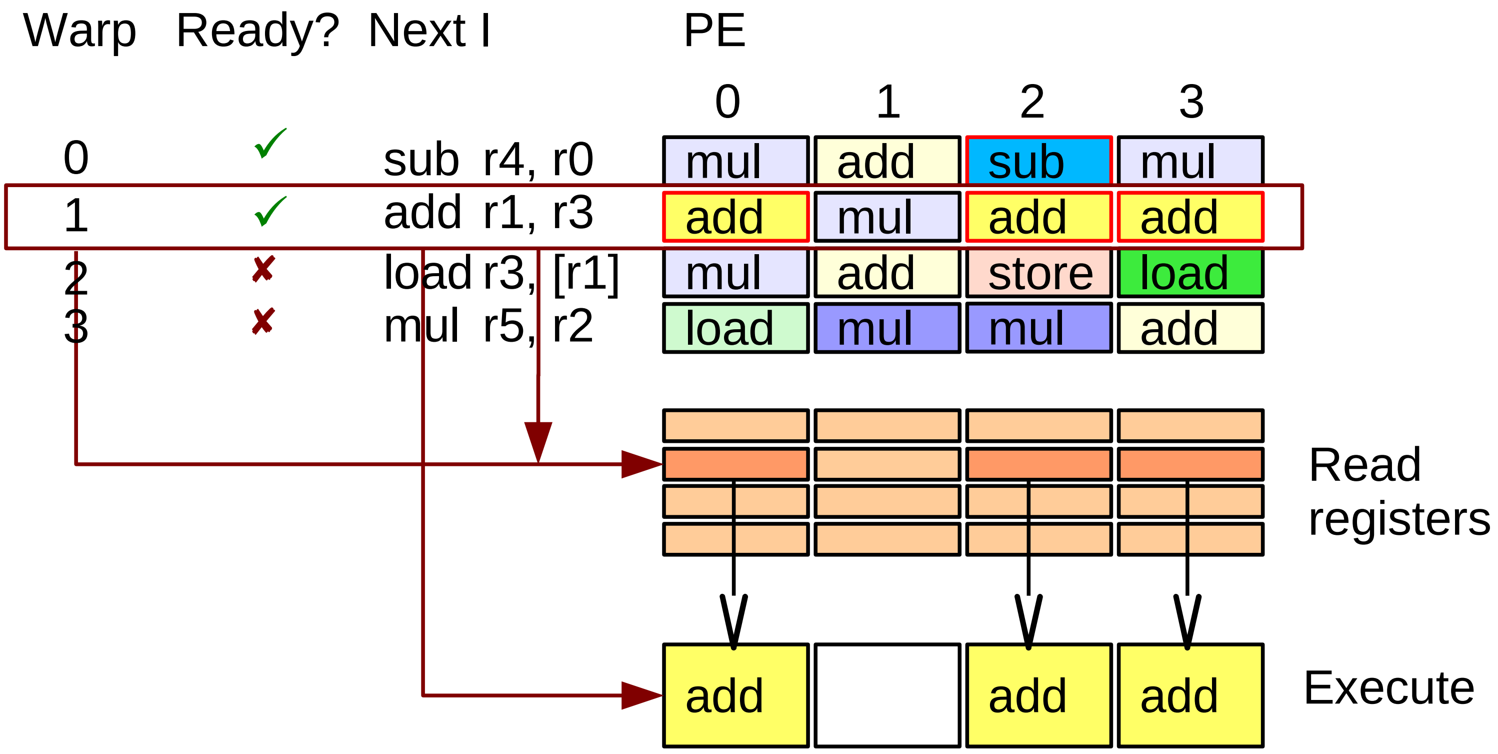
Benefits of multiple-PC arbitration

- Before: stack, counters
  - O(*n*), **O(log *n*)** memory
    n = nesting depth
  - **1 R/W port** to memory
  - **Exceptions**: stack overflow, underflow
- Still SIMD semantics (Bougé-Levaire)
  - Structured control flow only
  - Specific instruction sets

- After: multiple PCs
  - **O(1)** memory
  - **No shared state**
  - Allows thread **suspension, restart, migration**
- True SPMD semantics (multi-thread)
  - Traditional languages, compilers
  - Traditional instruction sets
- **Enables many new architecture ideas**

With multiple warps

- Two-stage scheduling
  - Select one warp
  - Select one instruction (MPC) for this warp



Warp	Ready?	Next I	PE 0	1	2	3
0	✓	sub r4, r0	mul	add	sub	mul
1	✓	add r1, r3	add	mul	add	add
2	✗	load r3, [r1]	mul	add	store	load
3	✗	mul r5, r2	load	mul	mul	add

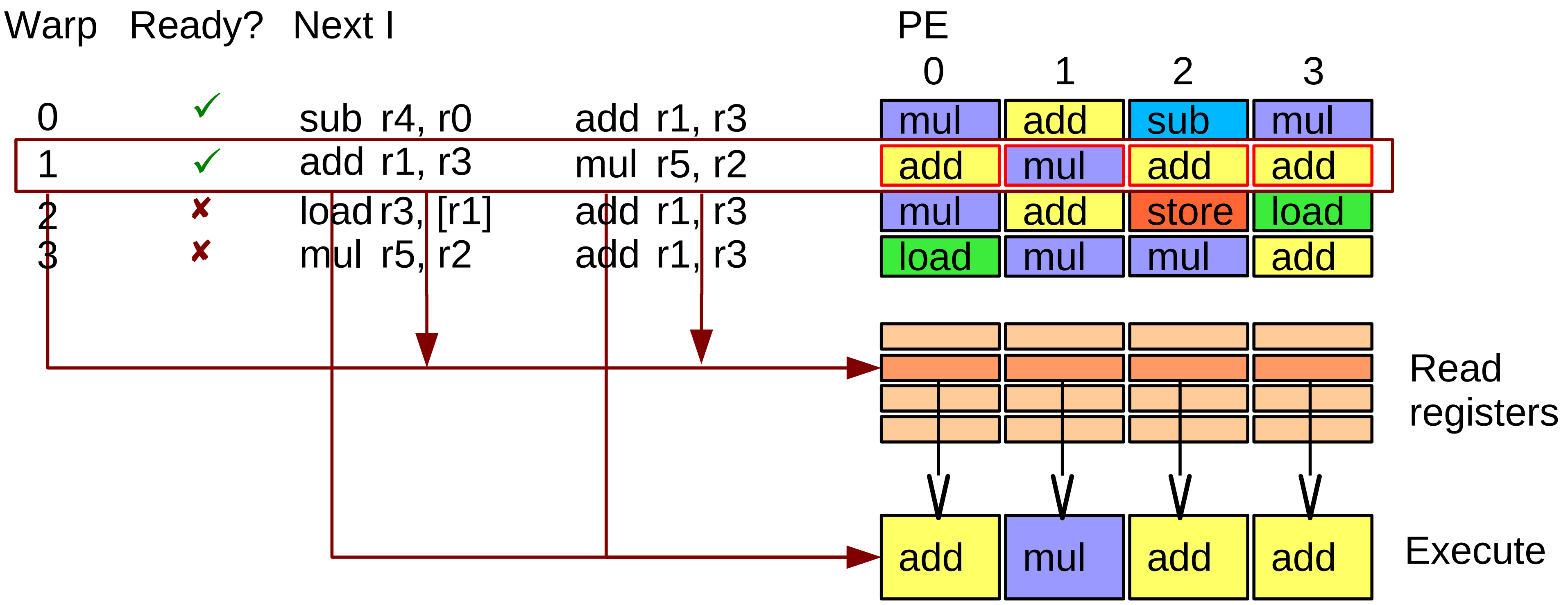
Read registers

Execute

add		add	add

Dual Instruction, Multiple Threads (DIMT)

- Two-stage scheduling
  - Select one warp
  - Select **two** instructions (MPC_1, MPC_2) for this warp



Warp	Ready?	Next I	
0	✓	sub r4, r0	add r1, r3
1	✓	add r1, r3	mul r5, r2
2	✗	load r3, [r1]	add r1, r3
3	✗	mul r5, r2	add r1, r3

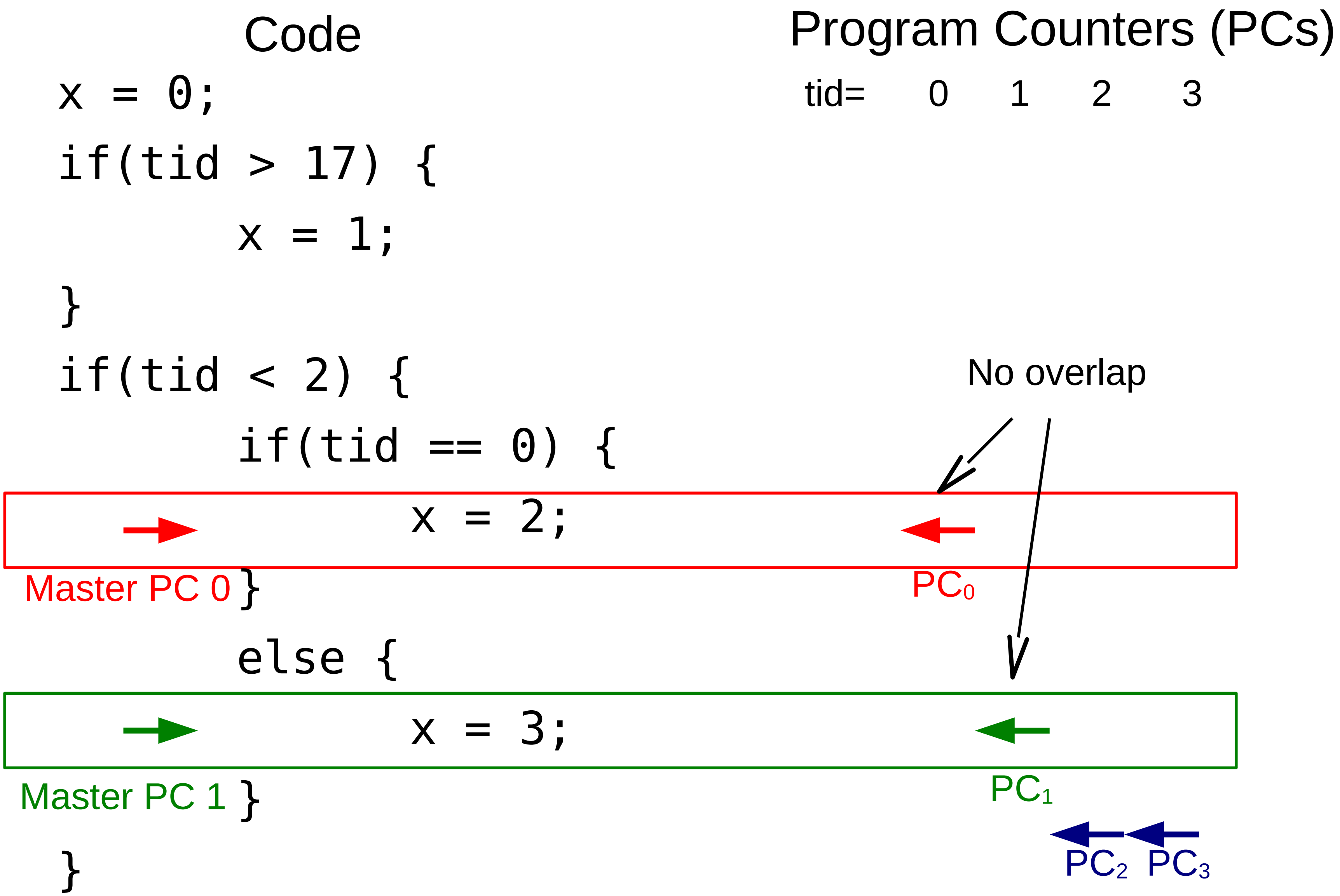
PE

0	1	2	3
mul	add	sub	mul
add	mul	add	add
mul	add	store	load
load	mul	mul	add

- More than 2 instructions: NIMT

A. Glew. *Coherent vector lane threading.* Berkeley ParLab Seminar, 2009.

Why DIMT?



Code

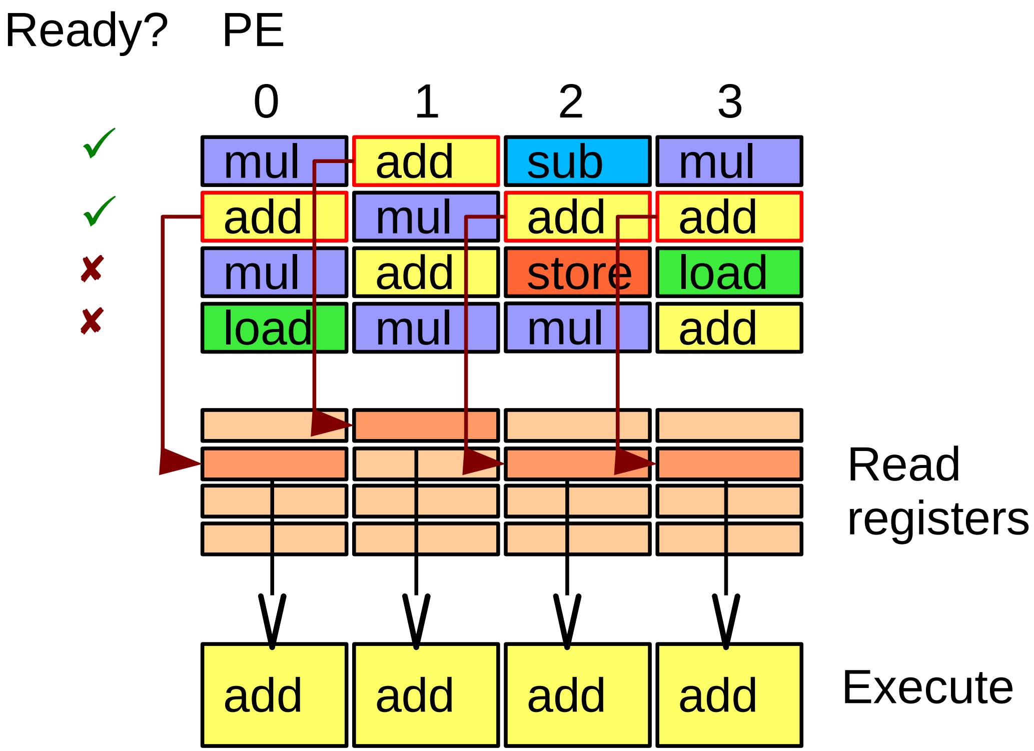
```
x = 0;
if(tid > 17) {
        x = 1;
}
if(tid < 2) {
        if(tid == 0) {
                x = 2;
        }
        else {
                x = 3;
        }
}
```

Program Counters (PCs)

- “Fills holes” using parallelism between execution paths

Dynamic Warp Formation (DWF)

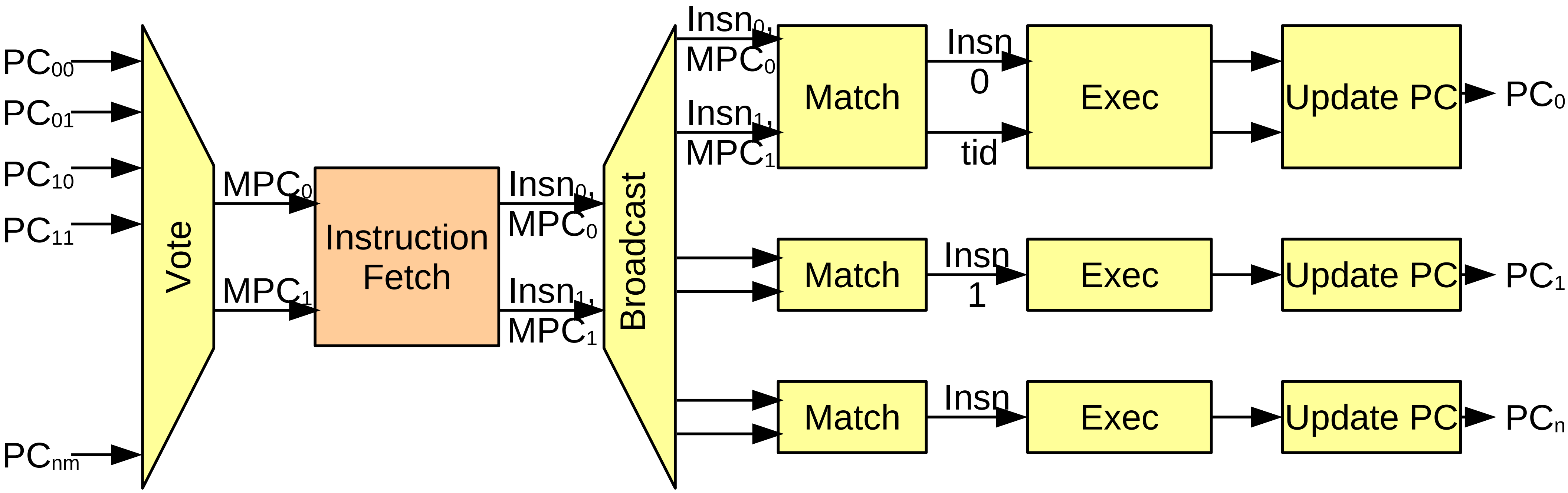
- Why need warps at all?
  - Select master PC from global thread pool
  - On each PE, select one thread from local thread pool



Ready?	PE 0	1	2	3
✓	mul	add	sub	mul
✓	add	mul	add	add
✗	mul	add	store	load
✗	load	mul	mul	add

W. Fung, I. Sham, G. Yuan, and T. Aamodt. *Dynamic warp formation and scheduling for efficient GPU control flow.* MICRO'07, 2007.

New DIMT+DWF pipeline



- Radical departure from classical SIMD

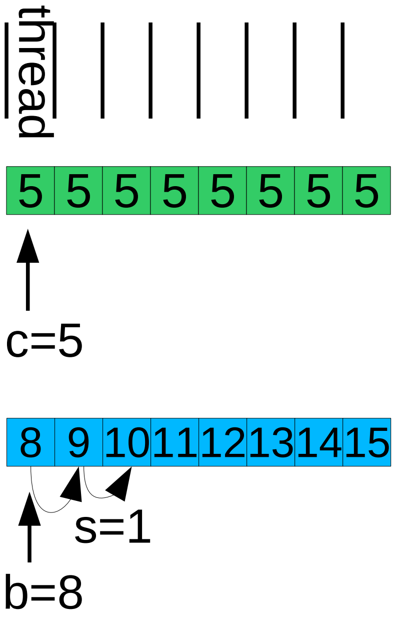
Avoiding redundancy

- Goal: keep execution units busy?

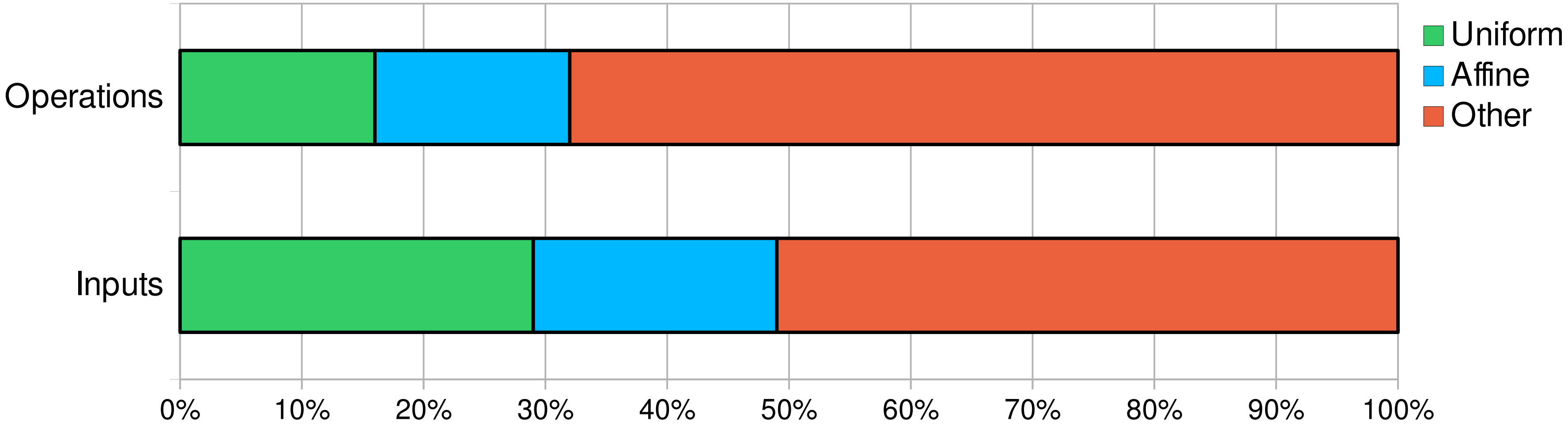
- Keep execution units busy **doing real work**!

What are we computing on?

- Uniform data
  - In a warp, v[*i*] = *c*
- Affine data
  - In a warp, v[*i*] = *b* + *i* *s*
  - Base *b*, stride *s*



- Average frequency on GPGPU applications

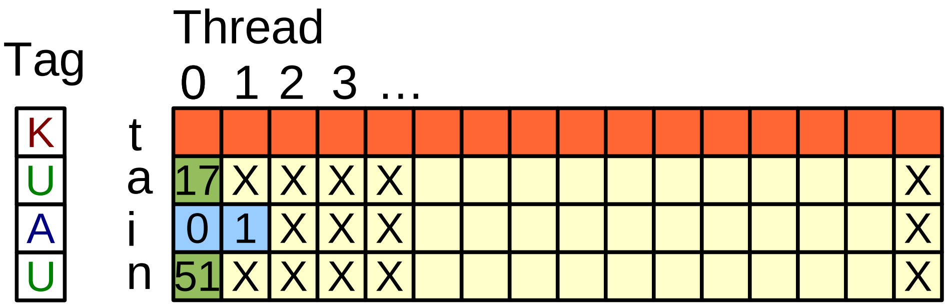


Tagging registers

- Associate a tag to each vector register
  - **U**niform, **A**ffine, un**K**nown
- Propagate tags across arithmetic instructions
- 2 lanes are enough to encode uniform and affine vectors

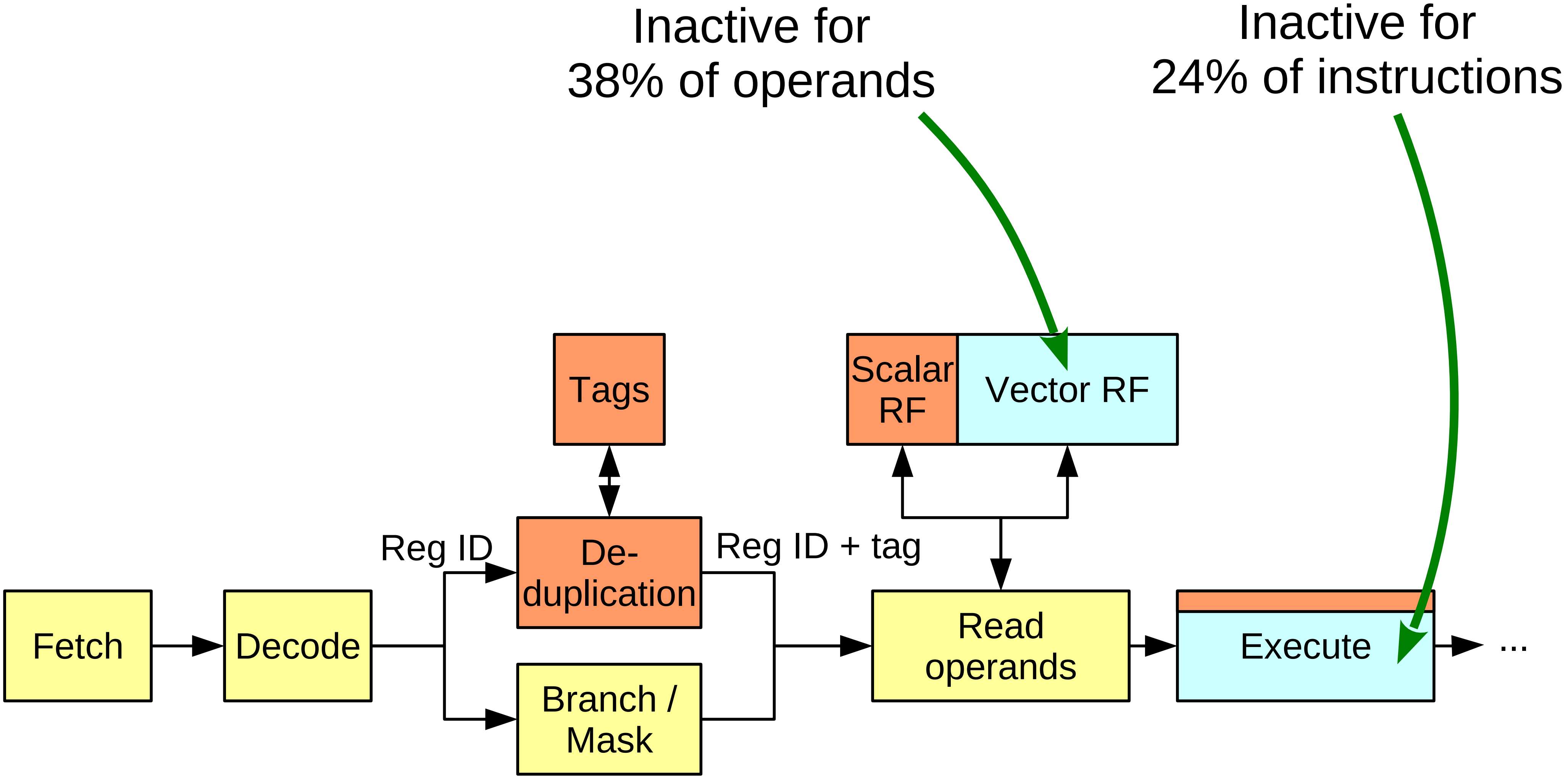
Trace

```
            Instructions              Tags
    mov    i ← tid                A←A
loop:
    load   t ← X[i]               K←U[A]
    mul    t ← a×t                K←U×K
    store X[i] ← t                U[A]←K
    add    i ← i+tcnt             A←A+U
    branch i<n? loop              A<U?
loop:
    load   t ← X[i]               K←U[A]
    mul    t ← a×t                K←U×K
    ...
```



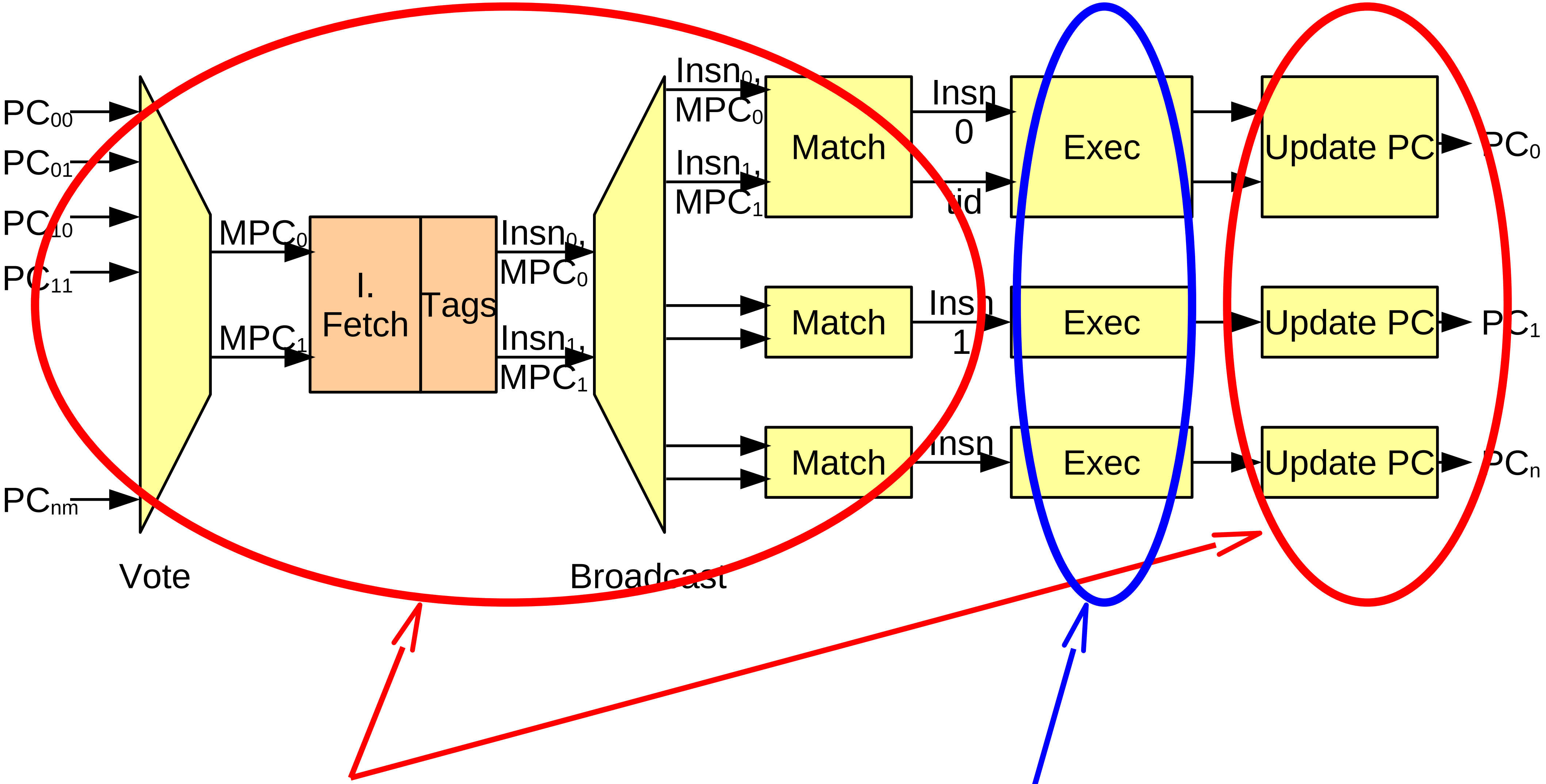
Tag		Thread 0	1	2	3	…											
K	t																
U	a	17	X	X	X	X											X
A	i	0	1	X	X	X											X
U	n	51	X	X	X	X											X

Dynamic Work Factorization ~~(DWF)~~



C. Collange, D. Defour, Y. Zhang. *Dynamic detection of uniform and affine vectors in GPGPU computations.* Europar HPPC09, 2009

Catch-22



- **Control logic** needs to stay **much** smaller / simpler / less power-hungry than **Execution logic**
- Is execution unit utilization such an issue anyway?

Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Dealing with variable latency

It's the memory, stupid!

- Our primary constraint: power
- Power measurements on NVIDIA GT200

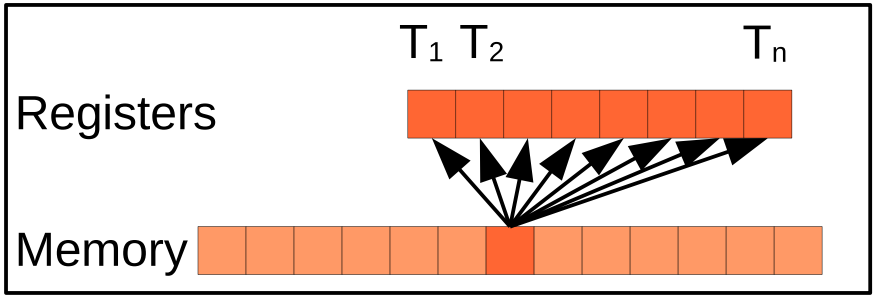
	Energy/op (nJ)	Total power (W)
Instruction control	1.8	18
Multiply-add on a 32-wide warp	3.6	36
Load 128B from DRAM	80	90

- With the same amount of energy
  - Load 1 word from DRAM
  - Compute 44 flops
- Memory traffic is what matters (most)

C. Collange, D. Defour, A. Tisserand. *Power consumption of GPUs from a software perspective.* ICCS 2009.

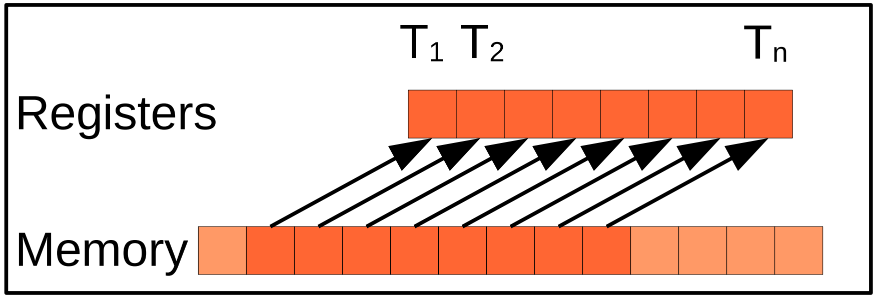
Memory access patterns

In traditional vector processing



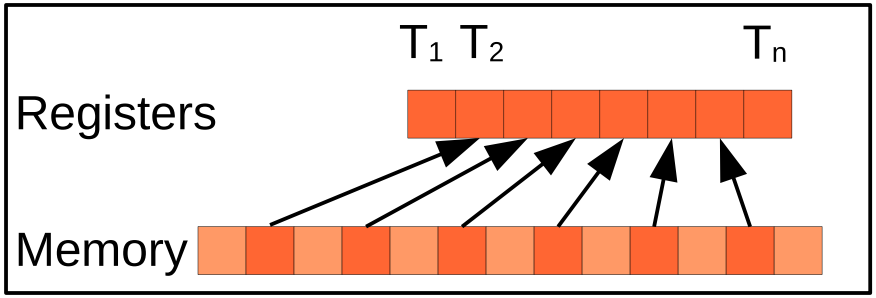
Scalar load & broadcast

Reduction & scalar store



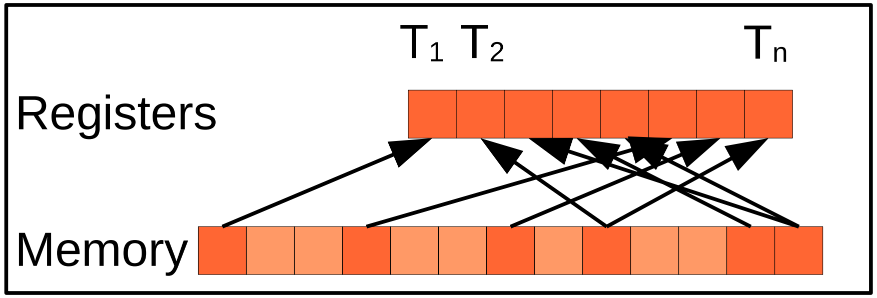
Unit-strided load

Unit-strided store



(Non-unit) strided load

(Non-unit) strided store



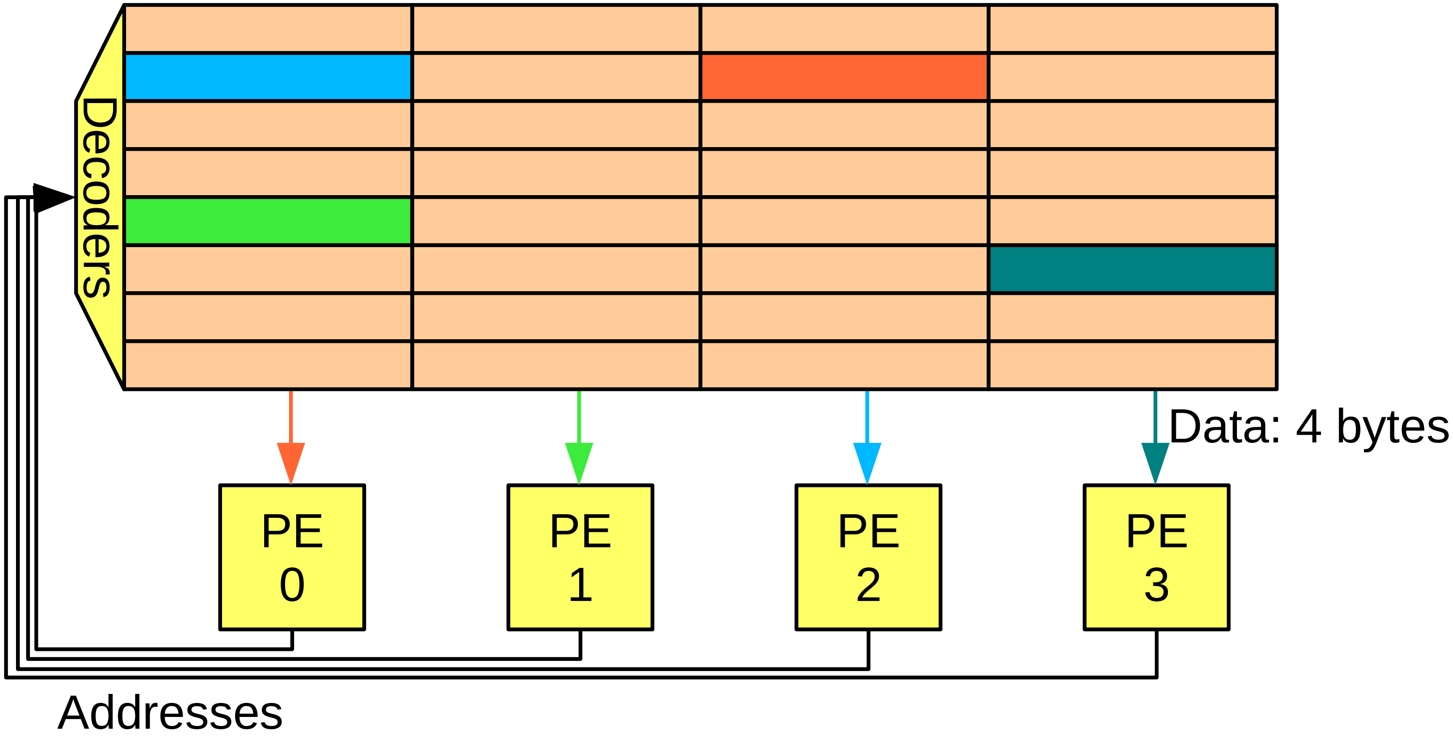
Gather

Scatter

In SIMT

- Every load is a gather, every store is a scatter

The memory we want



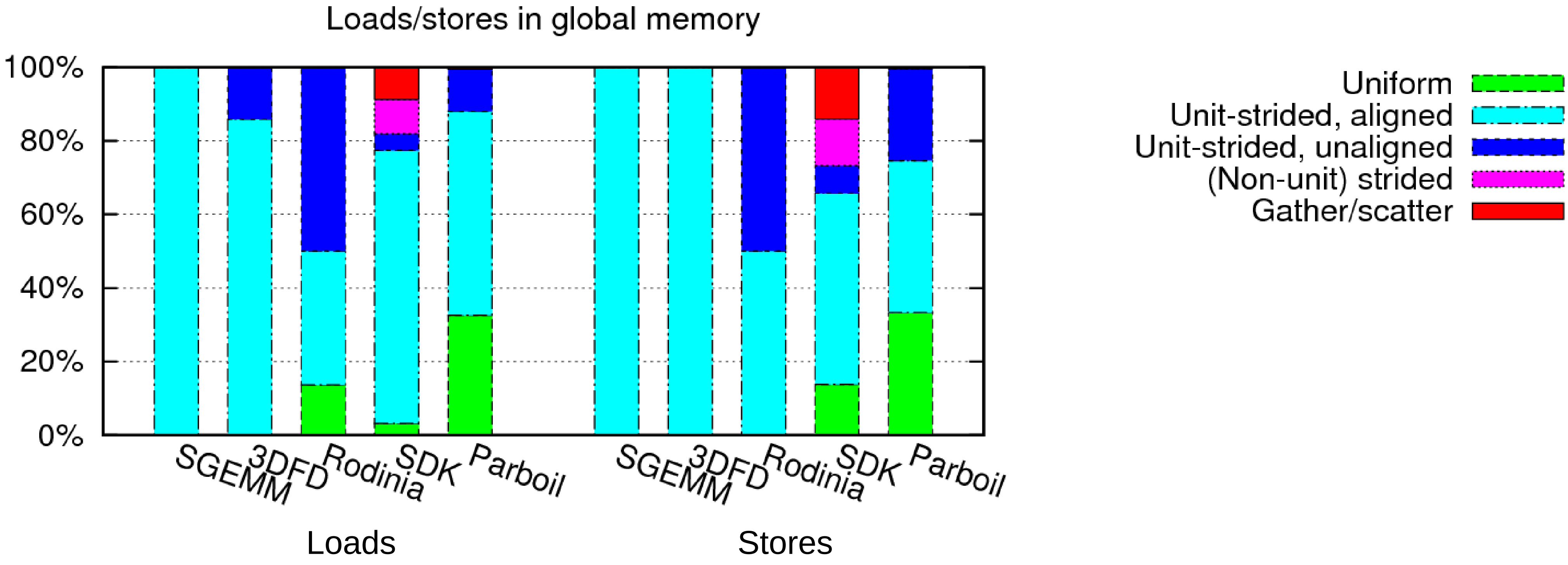
- Many independent R/W ports
- Supports lots of small transactions: 4B or 8B-wide

The memory we have

- DRAMs
  - Wide bus, burst mode
    - Use **wide transactions** (≤32B)
  - Switching DRAM pages is expensive
    - **Group** accesses by pages (1 page ∫ 2KB)
  - **One shared bus, r**ead/write turnaround penalty
    - **Group** accesses by direction
- Caches
  - Have wide cache lines (128B-256B)
  - Have **few R/W ports**

Breakdown of memory access patterns

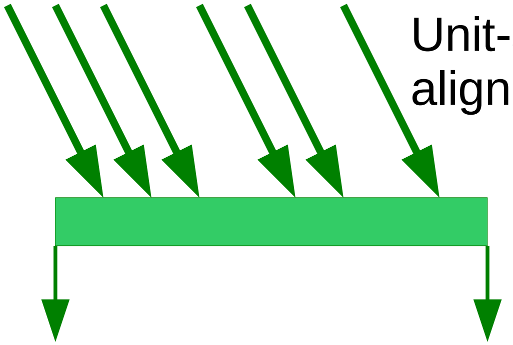
- Vast majority: uniform or unit-strided
  - And even aligned vectors



"*In making a design trade-off, favor the frequent case over the infrequent case.*" [HP06]

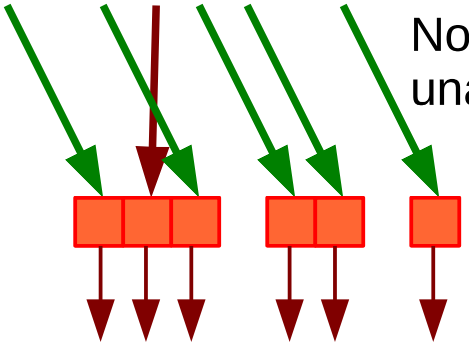
Coalescing concurrent requests

- Unit-strided detection (NVIDIA CC 1.0-1.1 coalescing)



Unit-strided and aligned requests

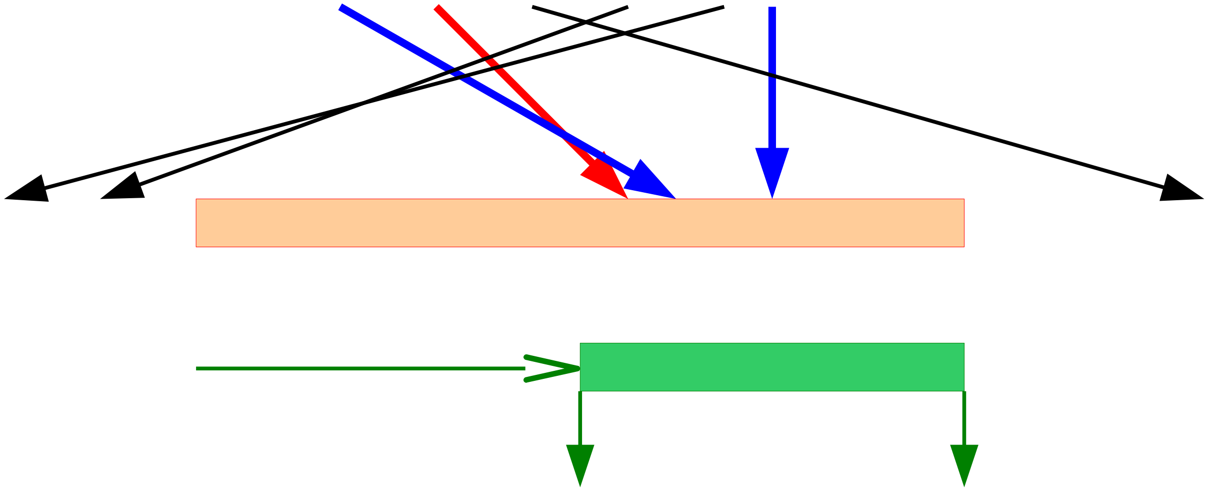
↑ One transaction



Non-unit-strided or unaligned requests

↑ Multiple transactions

- Minimal coverage (NVIDIA CC 1.2 coalescing)



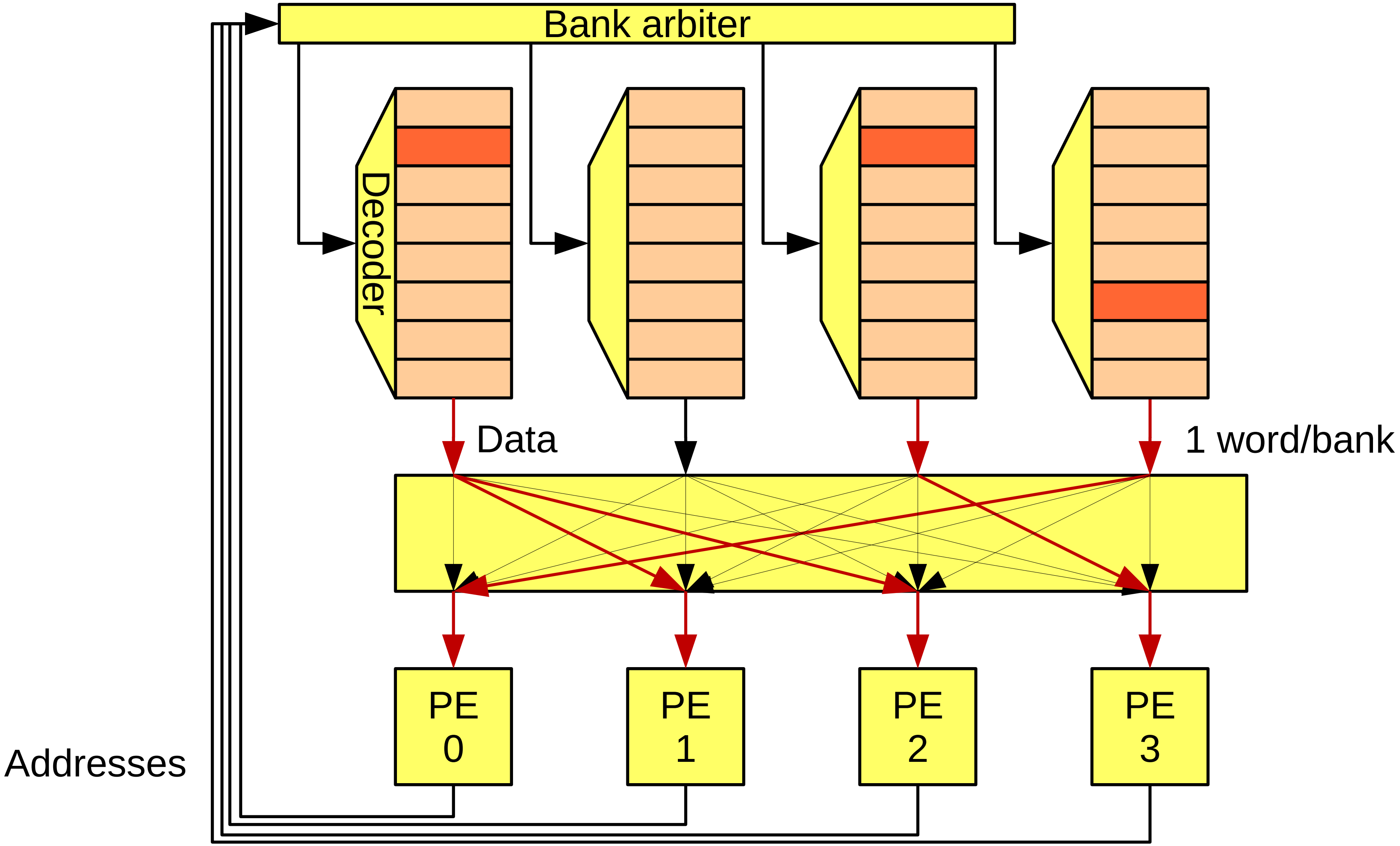
1. Select one request, consider maximal aligned transaction

2. Identify requests that fall in the same memory segment

3. Reduce transaction size when possible and issue transaction

4. Repeat with remaining requests

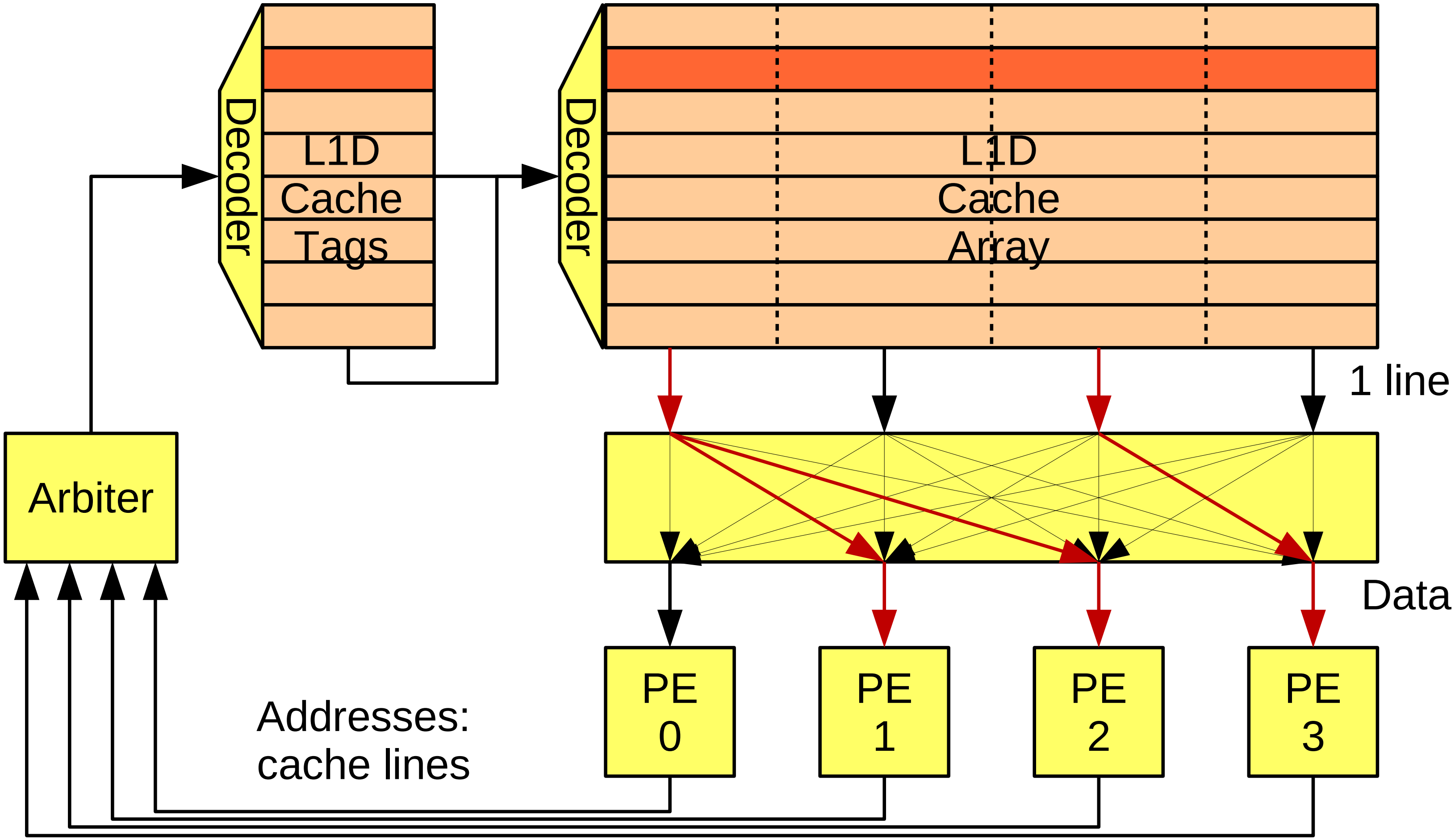
Banked shared memory



- Software-managed memory
- Interleaved on a word-by-word basis

Used in NVIDIA Tesla (2007)

Hardware-managed cache



- Share one wide port to the L1 cache
- Multiple lanes can read from the same cache line
- Bottleneck: single-ported cache tags

Used in NVIDIA Fermi (2010)

Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Dealing with variable latency

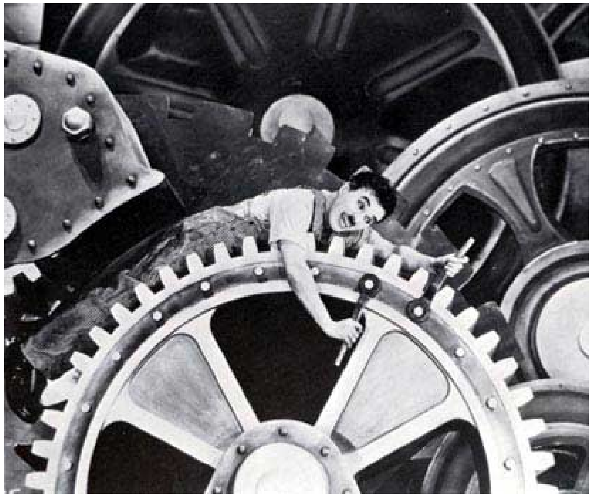
Dealing with pipeline hazards

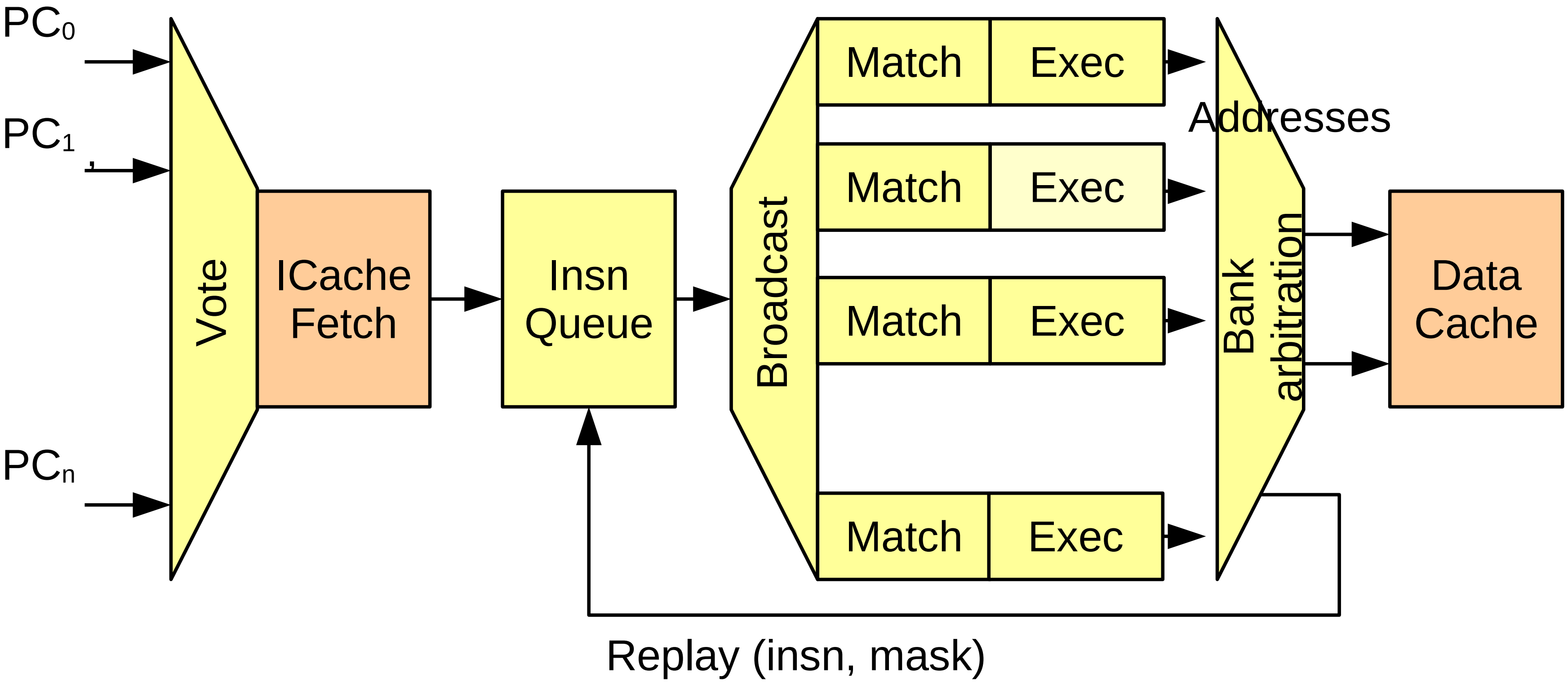
- Bank conflicts
- Lost arbitration
- Cache misses

- Conventional solution: stall execution pipeline until resolved

Preferred solution: in-order replay

- Instruction replay
  - Keep pipeline running
  - Put back offending instruction in instruction queue
  - With updated pred mask: only replay threads that failed





Used in NVIDIA Tesla (2007)

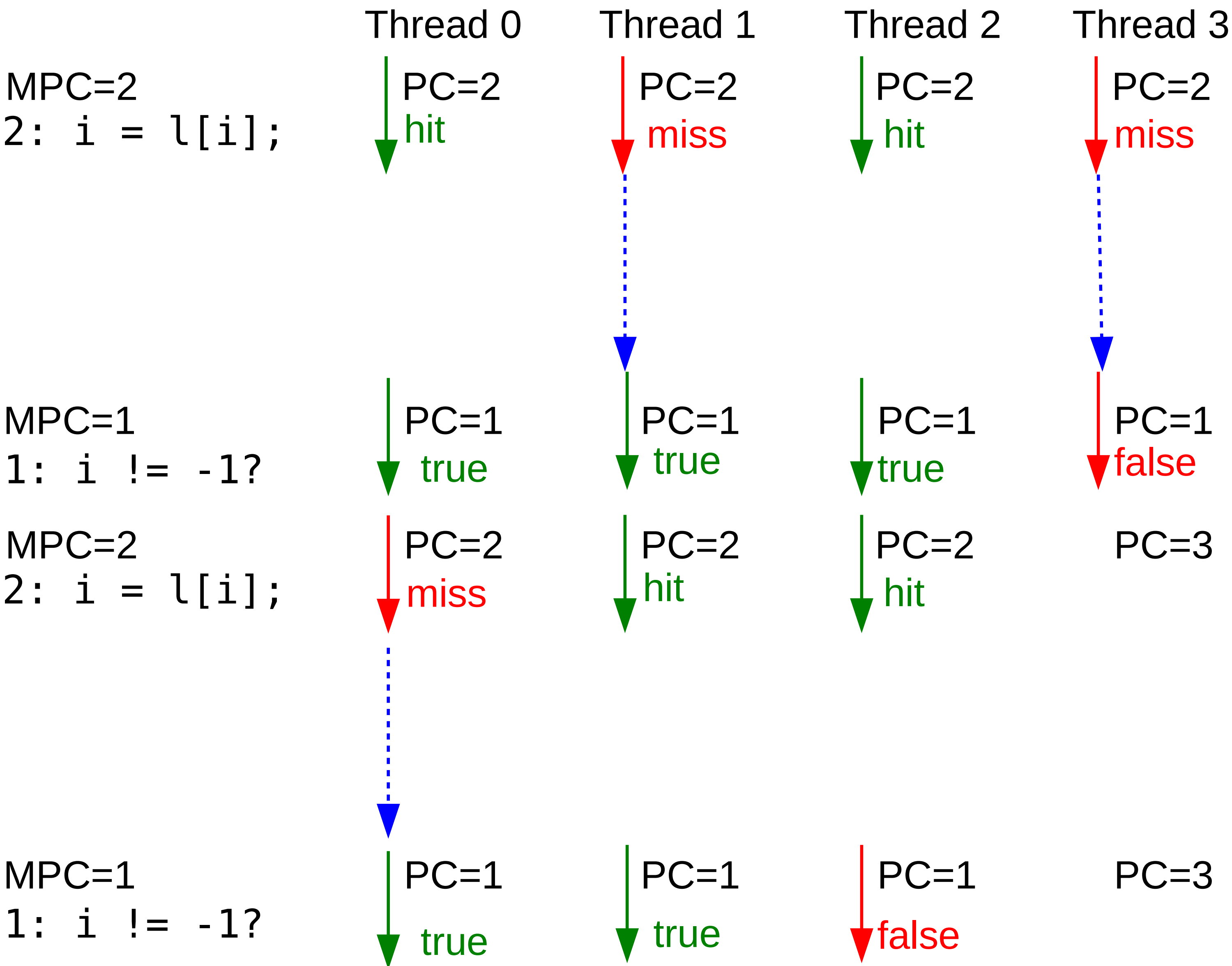
Dynamic Warp Subdivision

- Consider Replay as a control-flow operation (or no-op)
  - Threads that miss are turned inactive until data arrives
  - Threads that hit ask for next instruction
- Memory divergence = branch divergence
  - Both handled the same way
- When one thread misses, no need to block the whole warp
- Tradeoff: more latency hiding, lower ALU utilization
  - Could counteract utilization loss with DIMT/NIMT?

J. Meng, D. Tarjan and K. Skadron. *Dynamic warp subdivision for integrated branch and memory divergence tolerance.* ISCA'2010, 2010.

Linked list traversal: without DWS

```
1: while(i != -1) {
2:    i = l[i];
3: }
```



	Thread 0	Thread 1	Thread 2	Thread 3

MPC=2
2: i = l[i];
	PC=2
	hit
		PC=2
		miss
			PC=2
			hit
				PC=2
				miss

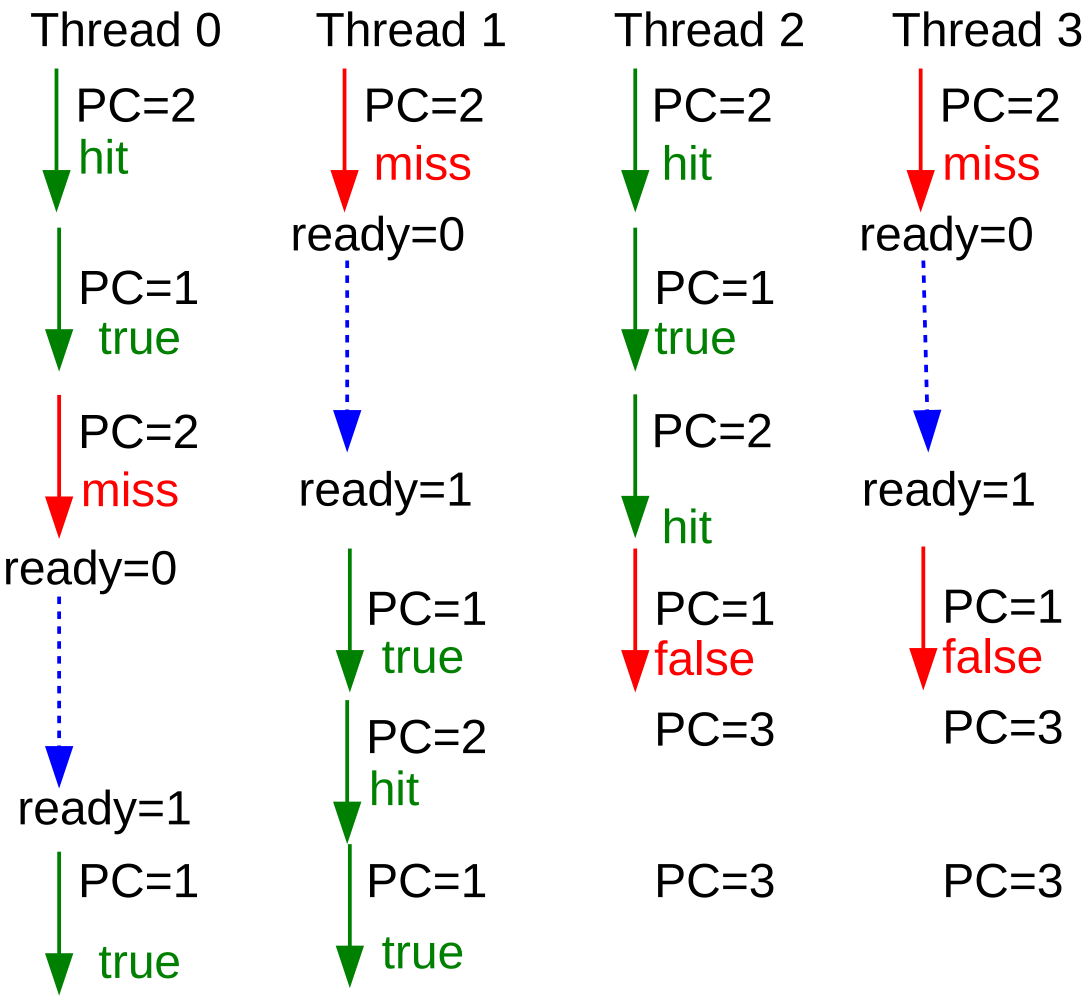
MPC=1 1: i != -1?	PC=1 true	PC=1 true	PC=1 true	PC=1 false

MPC=2
2: i = l[i];
	PC=2
	miss
		PC=2
		hit
			PC=2
			hit
				PC=3

MPC=1 1: i != -1?	PC=1 true	PC=1 true	PC=1 false	PC=3

Linked list traversal: with DWS

```
1: while(i != -1) {
2:    i = l[i];
3: }
```



	Thread 0	Thread 1	Thread 2	Thread 3

MPC=2
`2: i = l[i];`
	PC=2
	hit
		PC=2
		miss
		ready=0
			PC=2
			hit
				PC=2
				miss
				ready=0

MPC=1
`1: i != -1?`
	PC=1
	true
			PC=1
			true	

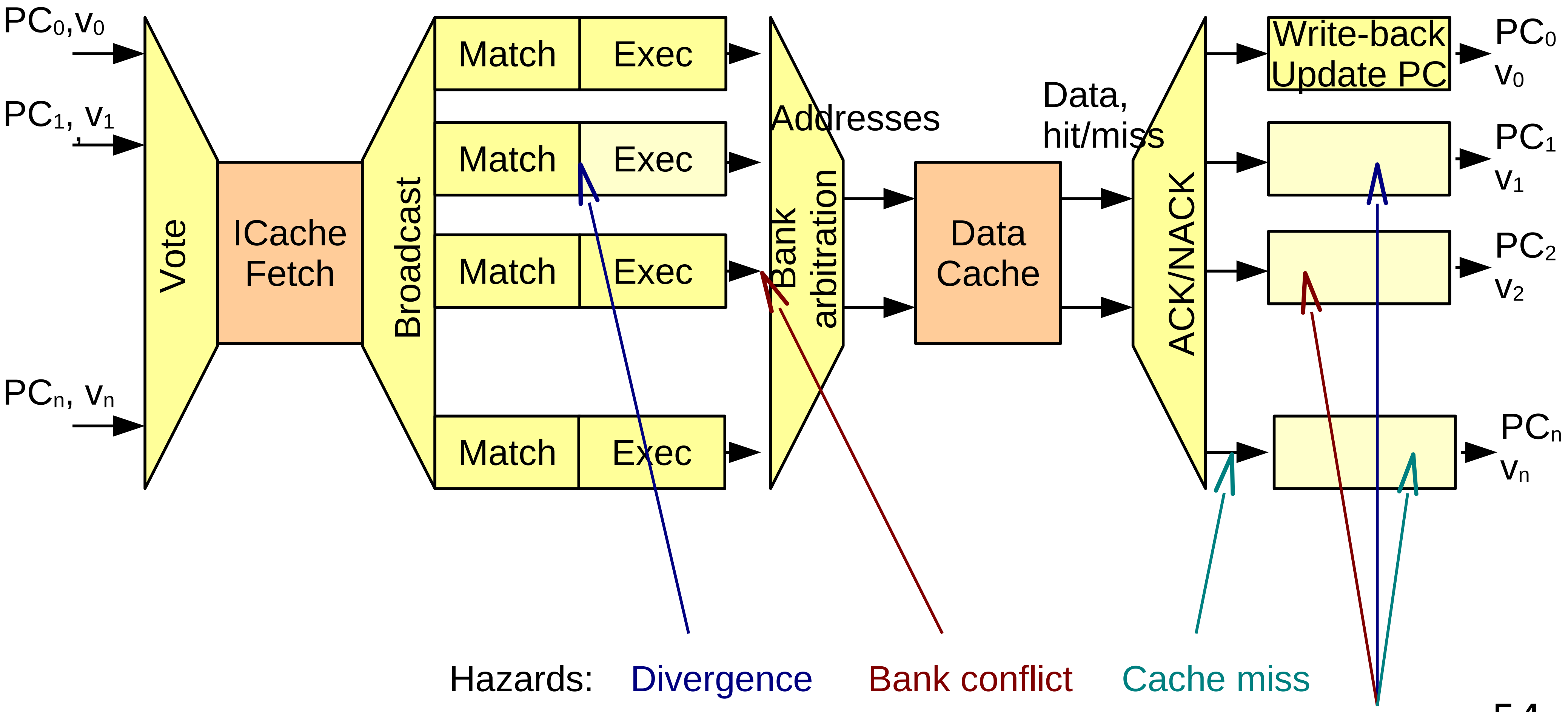
MPC=2
`2: i = l[i];`
	PC=2
	miss
	ready=0
		ready=1	PC=2
			hit
				ready=1

MPC=1
`1: i != -1?`
		PC=1
		true
			PC=1
			false
				PC=1
				false

MPC=2
`2: i = l[i];`
	ready=1	PC=2
		hit
			PC=3	PC=3

MPC=1
`1: i != -1?`
	PC=1
	true
		PC=1
		true
			PC=3	PC=3

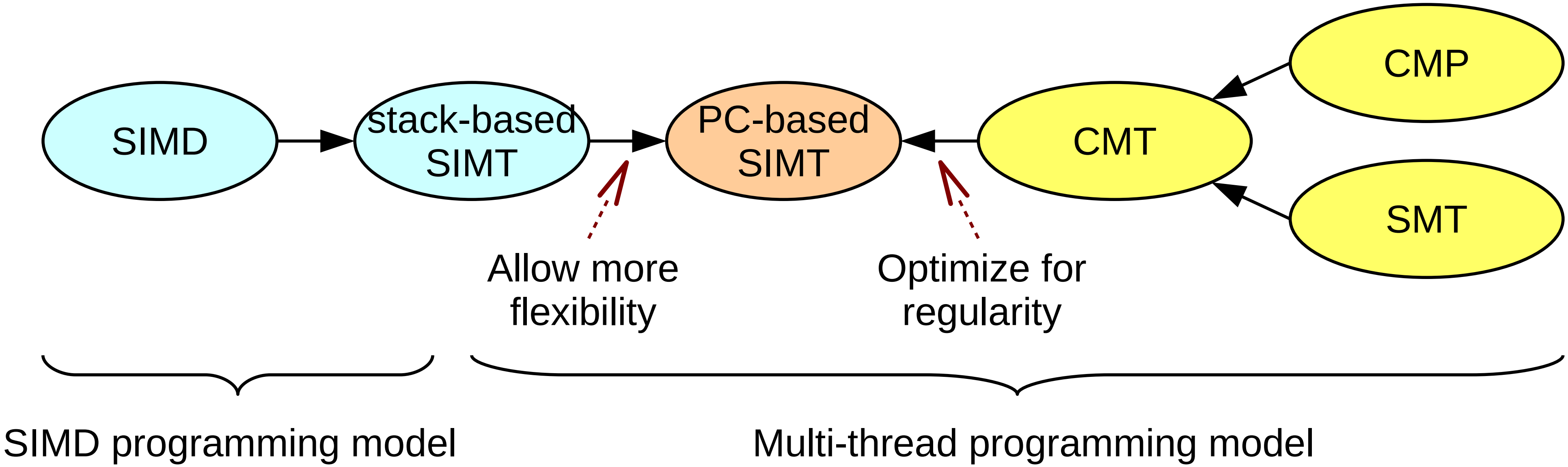
SIMT pipeline – memory instruction



Hazards: Divergence Bank conflict Cache miss

all cause PC and valid bit to be updated accordingly

Conclusion: the missing link



- New range of architecture options between Simultaneous Multi-Threading, Chip Multi-Threading and SIMD
  - Exploits parallel regularity for higher perf/W

Perspectives: next challenges

- Instruction fetch policy, thread scheduling policy: objectives to balance
  - Instruction throughput
  - Memory-level parallelism
  - Fairness
  - Regularity — coherence
- Detect control-flow reconvergence points
- Cross-fertilization with ideas from “classical” multi-threaded microarchitecture ?

Multi-threading or SIMD?
How GPU architectures exploit regularity

Caroline Collange
Arénaire, LIP, ENS de Lyon
caroline.collange@inria.fr

ARCHI'11
June 14, 2011