
Simty: Generalized SIMT execution on RISC-V

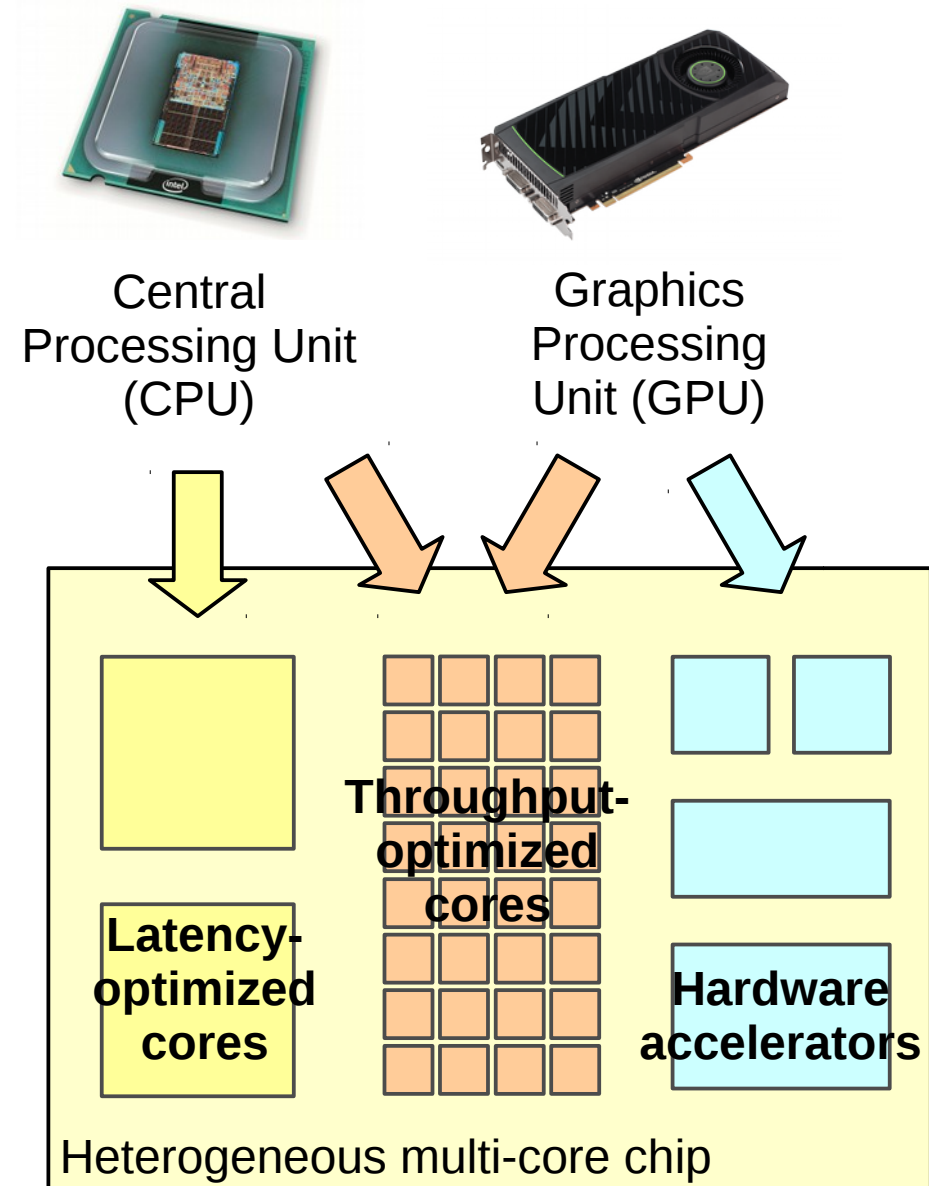
CARRV 2017

Sylvain Collange
INRIA Rennes / IRISA
sylvain.collange@inria.fr



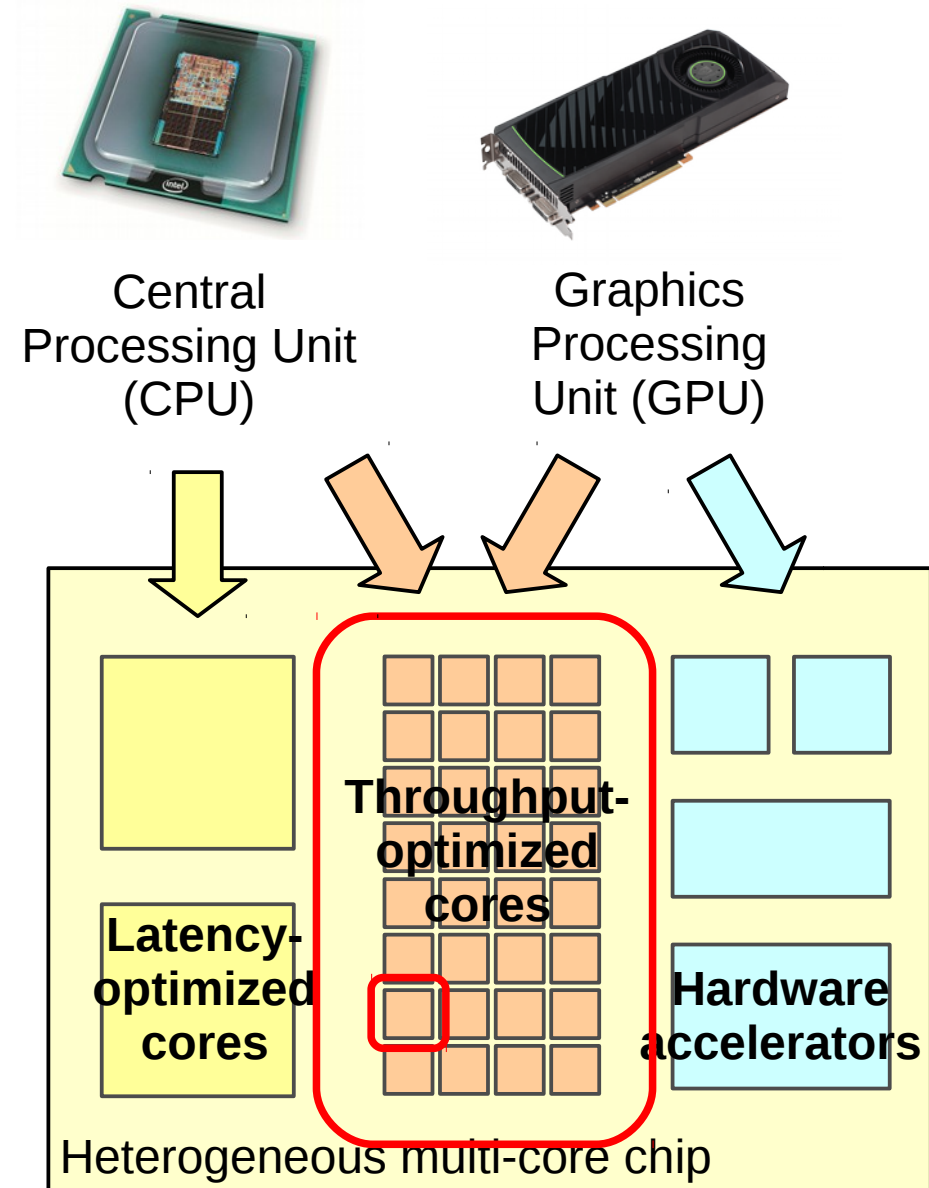
From CPU-GPU to heterogeneous multi-core

- Yesterday (2000-2010)
 - ◆ Homogeneous multi-core
 - ◆ Discrete components
- Today (2011-...)
Heterogeneous multi-core
 - ◆ **Physically unified**
CPU + GPU on the same chip
 - ◆ **Logically separated**
Different programming models, compilers, instruction sets
- Tomorrow
 - ◆ Unified programming models?
 - ◆ **Single instruction set?**



From CPU-GPU to heterogeneous multi-core

- Yesterday (2000-2010)
 - ◆ Homogeneous multi-core
 - ◆ Discrete components
- Today (2011-...)
Heterogeneous multi-core
 - ◆ **Physically unified**
CPU + GPU on the same chip
 - ◆ **Logically separated**
Different programming models, compilers, instruction sets
- Tomorrow
 - ◆ Unified programming models?
 - ◆ **Single instruction set?**
- **Defining the general-purpose throughput-oriented core**



Outline

- Stateless dynamic vectorization
 - ◆ Functional view
 - ◆ Implementation options
- The Simty core
 - ◆ Design goals
 - ◆ Micro-architecture

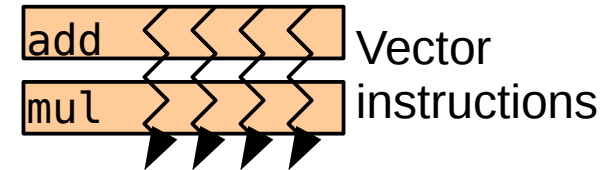
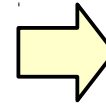
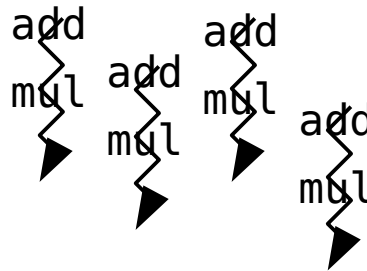
The enabler: dynamic inter-thread vectorization

- Idea: microarchitecture aggregates threads together to assemble vector instructions

SPMD Program

```
add r1, r3  
mul r2, r1
```

Threads



Vector
instructions

- ◆ Force threads to run in lockstep:
threads execute the same instruction at the same time (or do nothing)
- ◆ Generalization of GPU's SIMT for general-purpose ISAs
- ◆ **Benefits vs. static vectorization**
 - ◆ *Programmability*: software sees only threads, not threads + vectors
 - ◆ *Portability*: vector width is not exposed in the ISA
 - ◆ *Scalability*: + threads → larger vectors or more latency hiding or more cores
 - ◆ *Implementation simplicity*: handling traps is straightforward

Goto considered harmful?

RISC-V	NVIDIA Tesla (2007)	NVIDIA Fermi (2010)	Intel GMA Gen4 (2006)	Intel GMA SB (2011)	AMD R500 (2005)	AMD R600 (2007)	AMD Cayman (2011)
jal	bar	bar	jmp	jmp	jump	push	push
jalr	bra	bpt	if	if	loop	push_else	push_else
bXX	brk	bra	iff	else	endloop	pop	pop
ecall	brkpt	brk	else	endif	rep	loop_start	push_wqm
ebreak	cal	brx	endif	case	endrep	loop_start_no_al	pop_wqm
Xret	cont	cal	do	while	breakloop	loop_start_dx10	else_wqm
	kil	cont	while	break	breakrep	loop_end	jump_any
	pbk	exit	break	cont	continue	loop_continue	reactivate
	pret	jcal	cont	halt		loop_break	reactivate_wqm
	ret	jmx	halt	call		jump	loop_start
	ssy	kil	mshave	return		else	loop_start_no_al
	trap	pbk	mrest	fork		call	loop_start_dx10
	.s	pret	push			call_fs	loop_end
		ret	pop			return	loop_continue
		ssy				return_fs	loop_break
		.s				alu	jump
						alu_push_before	else
						alu_pop_after	call
						alu_pop2_after	call_fs
						alu_continue	return
						alu_break	return_fs
						alu_else_after	alu
							alu_push_before
							alu_pop_after
							alu_pop2_after
							alu_continue
							alu_break
							alu_else_after

Control transfer instructions in GPU instruction sets vs. RISC-V

- GPUs: control flow divergence and convergence is explicit
 - ◆ Incompatible with general-purpose instruction sets ☹️

Stateless dynamic vectorization

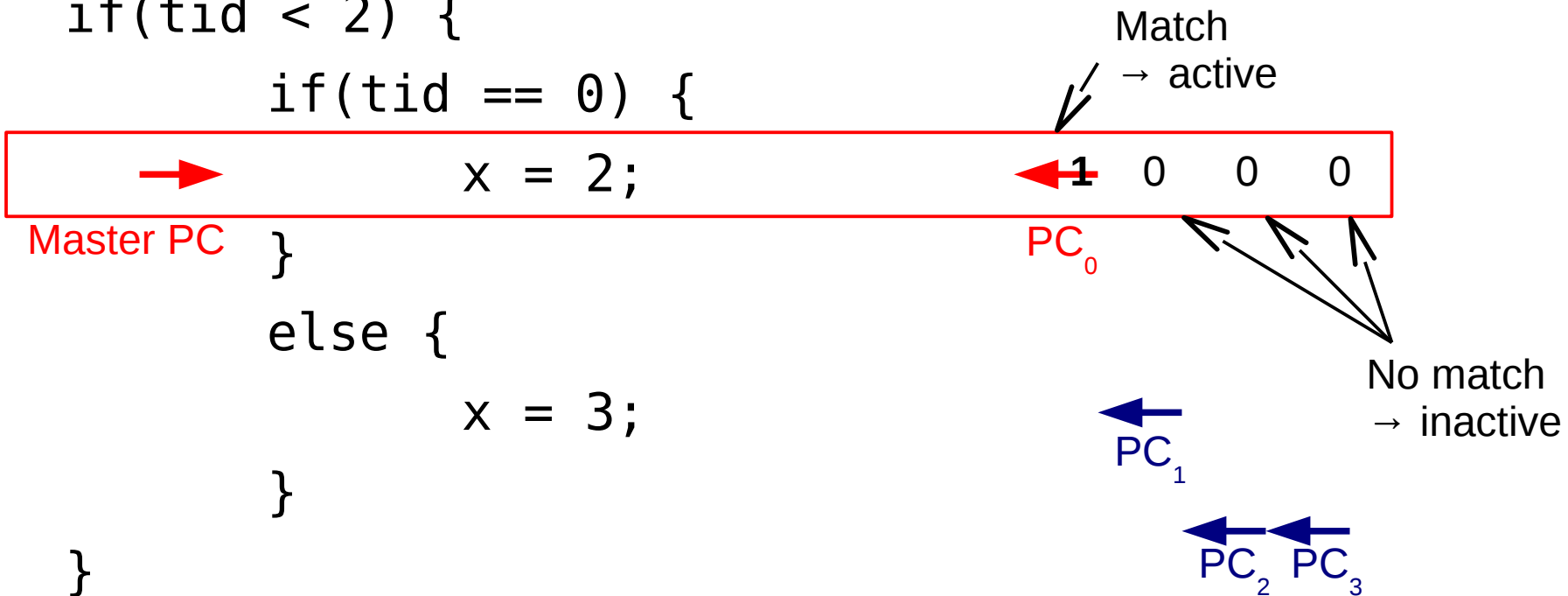
Idea: per-thread PCs characterize thread state

Code

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

Program Counters (PCs)

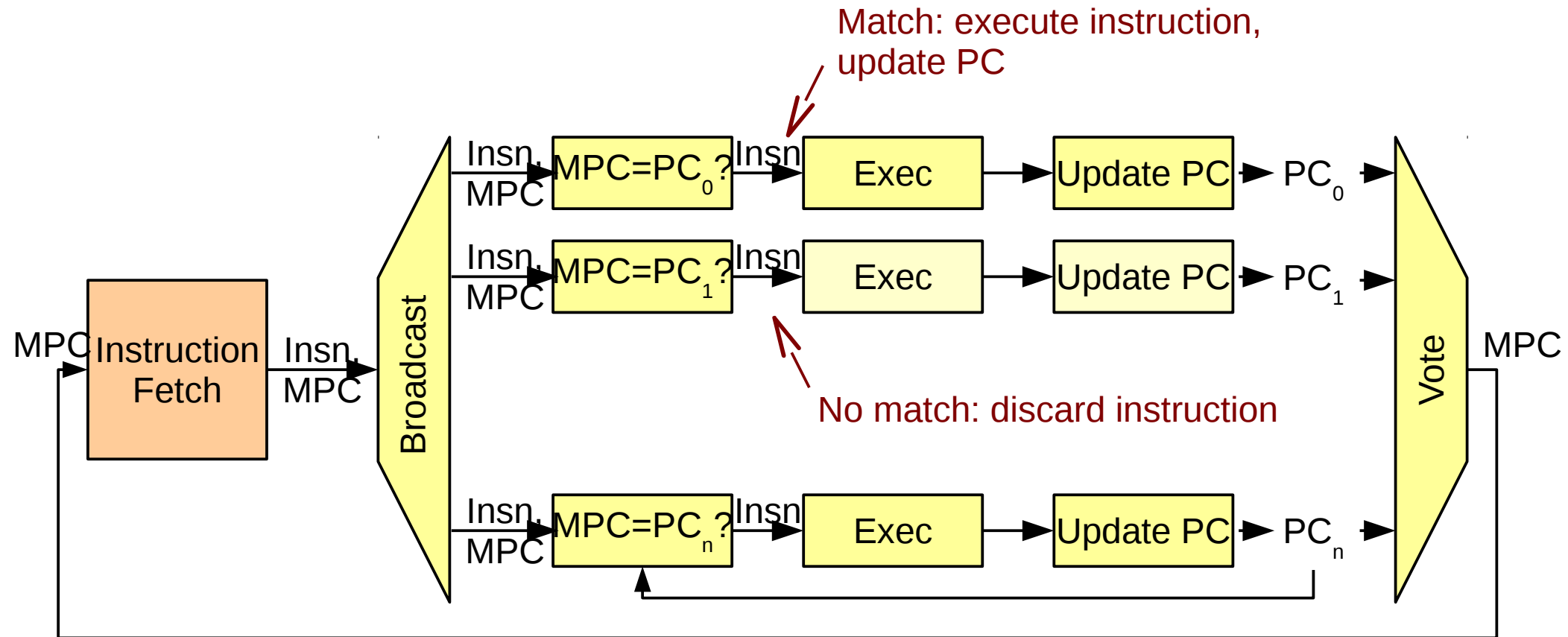
tid= 0 1 2 3



- Policy: MPC = $\min(PC_i)$ inside deepest function
 - ◆ Intuition: favor threads that are behind so they can catch up
 - ◆ Earliest reconvergence with code laid out in reverse post order

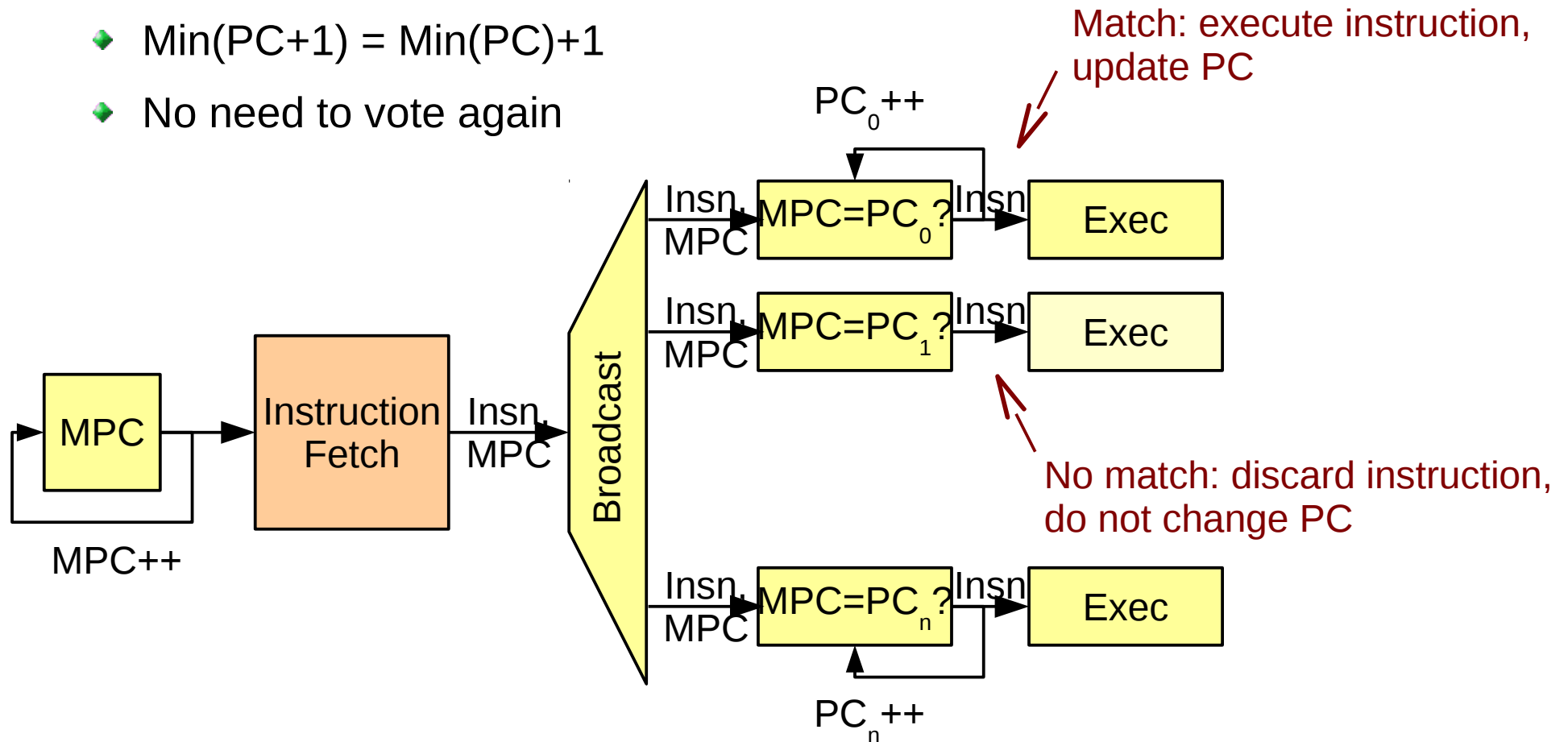
Functional view

- Control transfer instruction or exception



Functional view

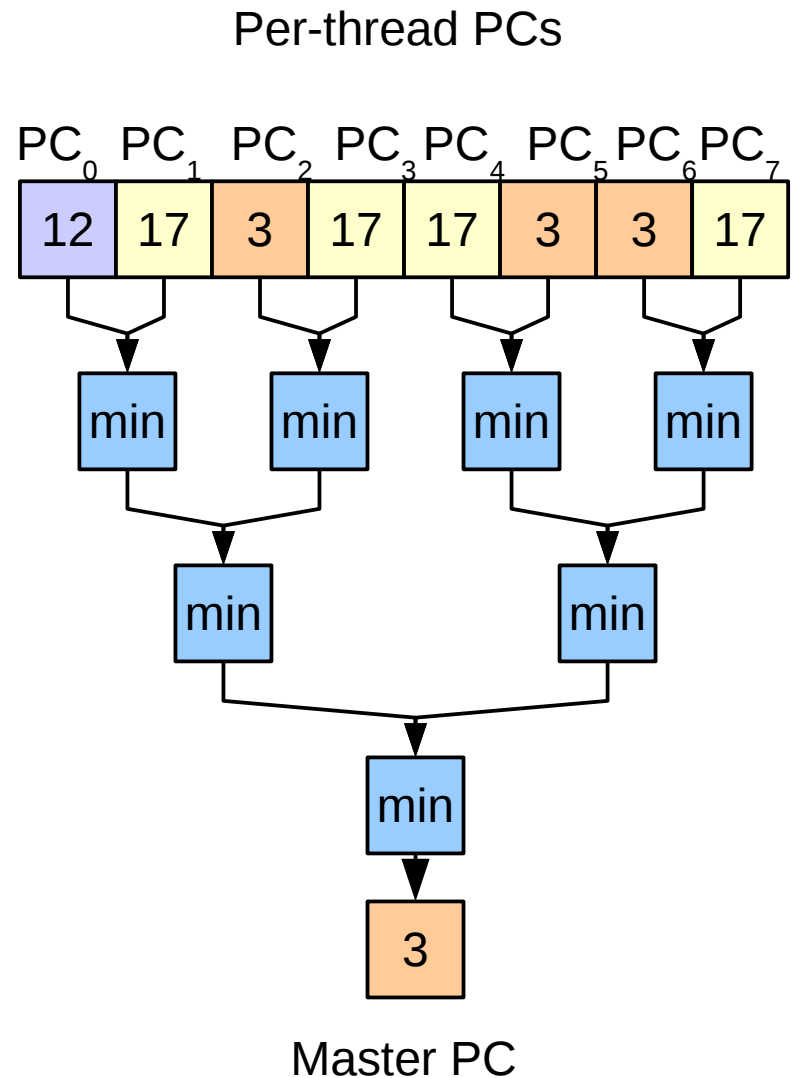
- Arithmetic instruction
 - ◆ $\text{Min}(\text{PC}+1) = \text{Min}(\text{PC})+1$
 - ◆ No need to vote again



Implementation 1: reduction tree

Straightforward implementation of the functional view

- On every branch: compute Master PC from individual PCs
 - ◆ Reduction tree to compute $\max(\text{depth})\text{-min}(\text{PCs})$
- On every instruction: compare Master PC with individual PCs
 - ◆ Row of address comparators
- Issues: area, energy overheads, extra branch resolution latency



Implementation 2: sorted context table

- Common case: few different PCs
- Order stable in time
- Keep Common PCs+activity masks in sorted heap

12	17	3	17	17	3	3	17
PC ₀	PC ₁	PC ₂	PC ₃	PC ₄	PC ₅	PC ₆	PC ₇

Per-thread PCs



		T ₀	T ₁					T ₇	
CPC ₁	3	0	0	1	0	0	1	1	0
CPC ₂	12	1	0	0	0	0	0	0	0
CPC ₃	17	0	1	0	1	1	0	0	1

Sorted context table

- Branch = insertion in sorted context table
- Convergence = fusion of head entries when CPC₁=CPC₂
- Activity mask is readily available

Outline

- Stateless dynamic vectorization
 - ◆ Functional view
 - ◆ Implementation options
- The Simty core
 - ◆ Design goals
 - ◆ Micro-architecture

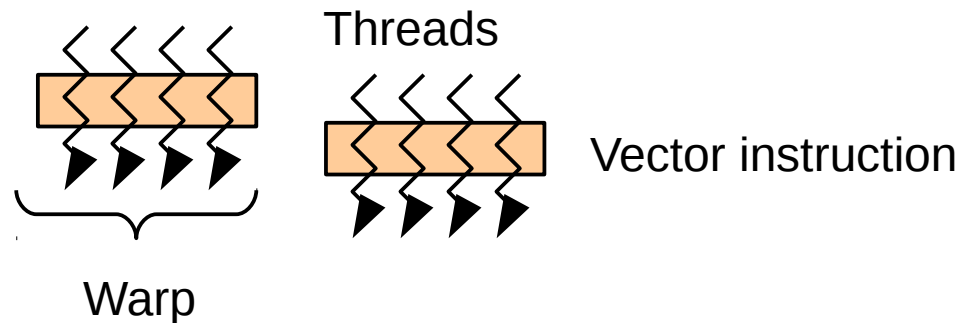
Simty: illustrating the simplicity of SIMT

Proof of concept for dynamic inter-thread vectorization

- Focus on the core ideas → the RISC of dynamic vectorization
- Simple programming model
 - ◆ Many scalar threads
 - ◆ General-purpose RISC-V ISA
- Simple micro-architecture
 - ◆ Single-issue RISC pipeline
 - ◆ SIMD execution units
- Highly concurrent, scalable
 - ◆ Interleaved multi-threading to hide latency
 - ◆ **Dynamic vectorization** to increase execution throughput
 - ◆ Target: hundreds of threads per core

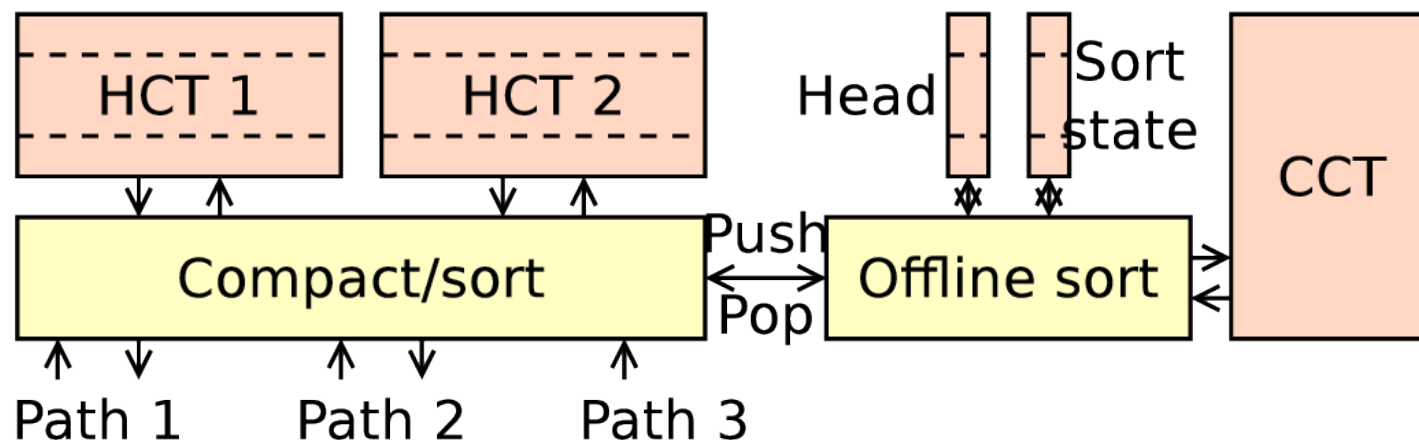
Multiple warps

- Wide dynamic vectorization found counterproductive
 - ◆ Sensitive to control-flow and memory divergence
 - ◆ Threads that hit in the cache wait for threads that miss
 - ◆ Breaks latency hiding capability of interleaved multi-threading
- Two-level approach : partition threads into warps, vectorize inside warps
 - ◆ Standard approach on GPUs



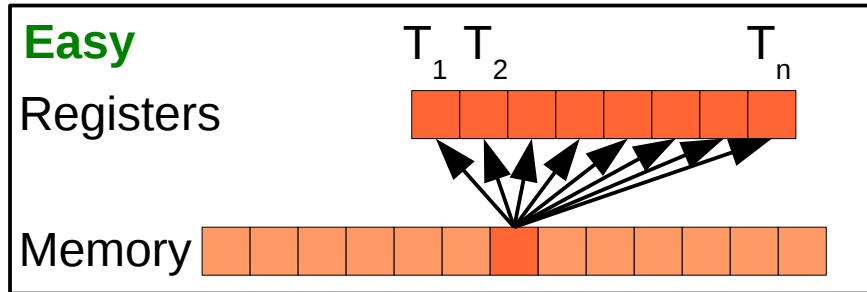
Two-level context table

- Cache top 2 entries in the *Hot Context Table* register
 - ◆ Constant-time access to CPC_i , activity masks
 - ◆ In-band convergence detection
- Other entries in the *Cold Context Table*
 - ◆ Branch → incremental insertion in CCT
 - ◆ Out-of-band CCT sorting: inexpensive insertion sort in $O(n^2)$
 - ◆ If CCT sorting cannot catch up: degenerates into a stack (=GPUs)

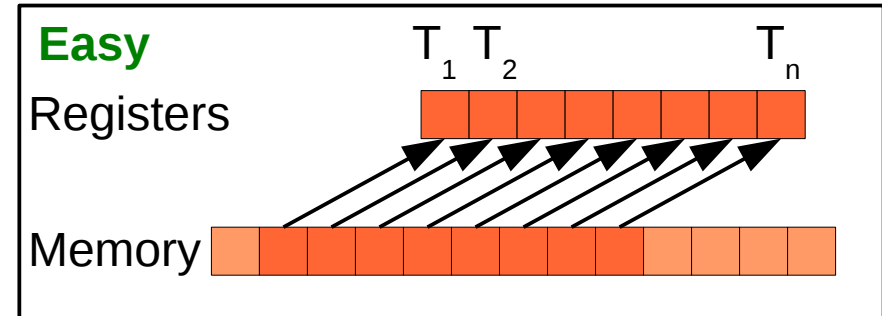


Memory access patterns

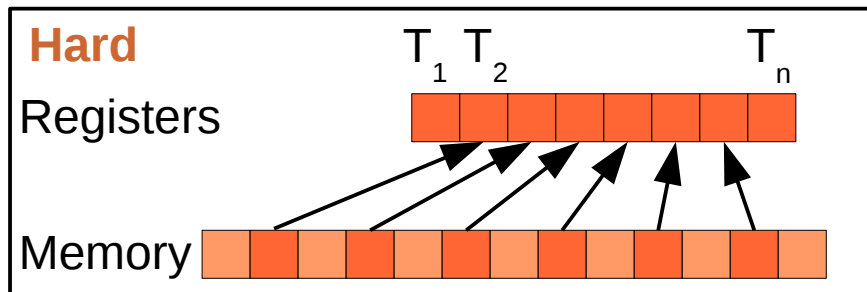
In traditional vector processing



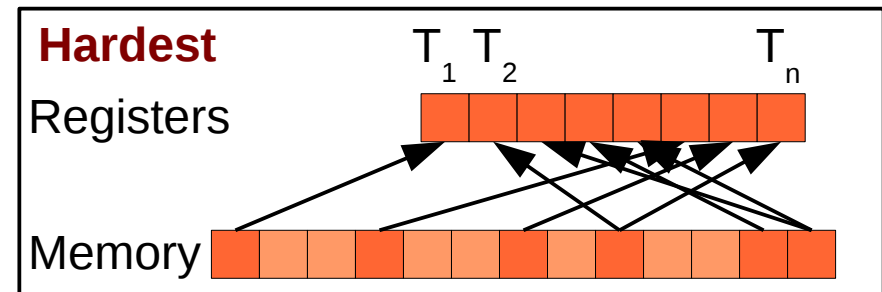
Scalar load & broadcast
Reduction & scalar store



Unit-strided load
Unit-strided store



(Non-unit) strided load
(Non-unit) strided store



Gather
Scatter

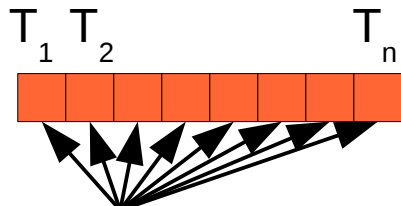
Memory access patterns

With dynamic vectorization

Easy

Registers

Memory



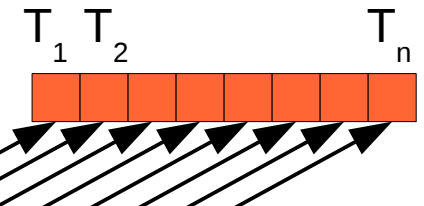
Scalar load & broadcast

Reduction & scalar store

Easy

Registers

Memory



Unit-strided load

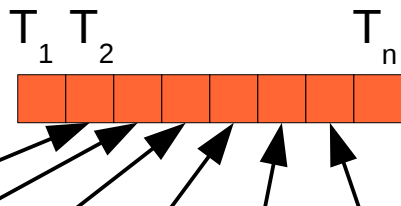
Unit-strided store

Common case

Hard

Registers

Memory



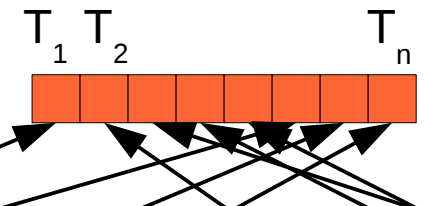
(Non-unit) strided load

(Non-unit) strided store

Hardest

Registers

Memory



Gather

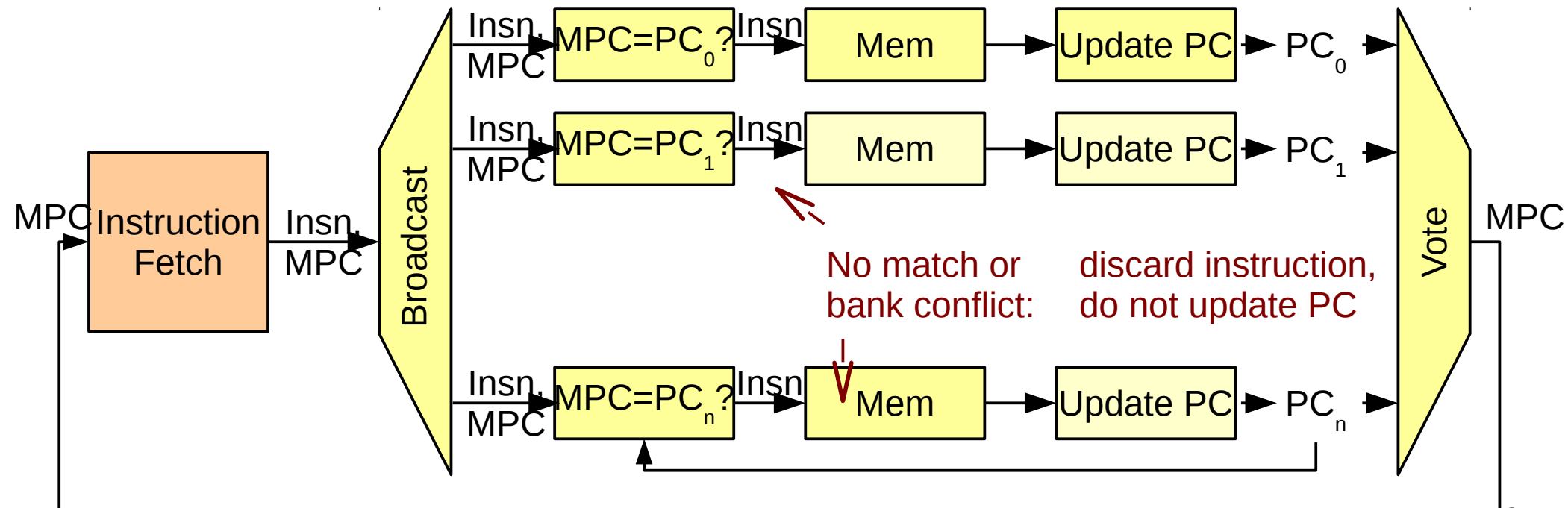
Scatter

General case

→ Support the general case, optimize for the common case

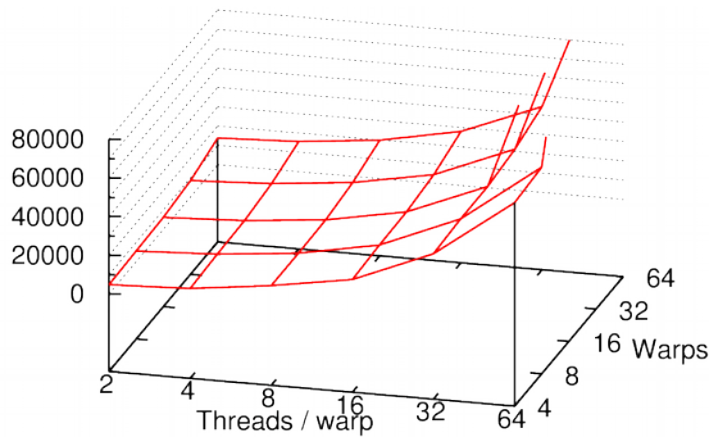
Memory access unit

- Scalar and aligned unit-strided scenarios: single pass
- Complex accesses in multiple passes using replay
- Execution of a scatter/gather is interruptible
 - ◆ Allowed by multi-thread ISA
 - ◆ No need to rollback on TLB miss or exception

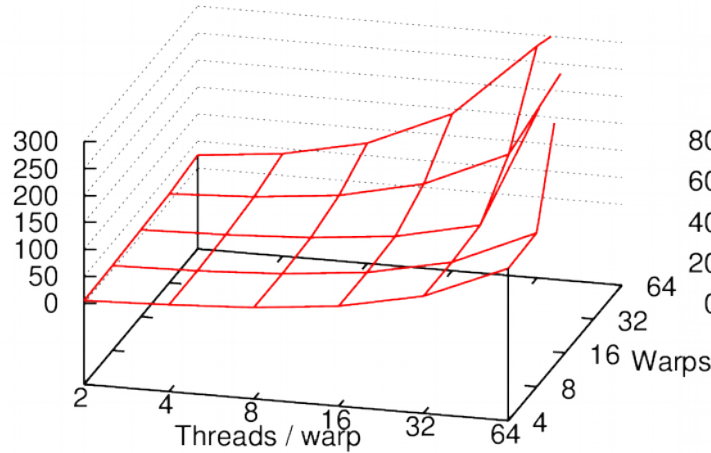


FPGA prototype

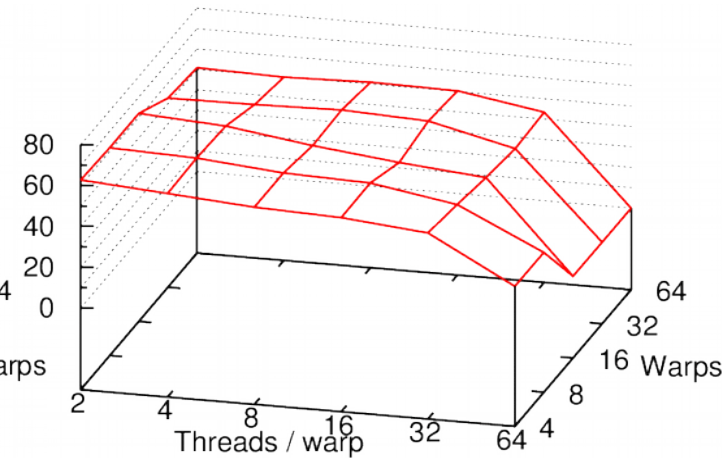
On Altera Cyclone IV



Logic area (LEs)



Memory area (M9Ks)



Frequency (MHz)

- Up to 2048 threads per core: 64 warps × 32 threads
- Sweet spot: 8x8 to 32x16

Latency hiding
multithreading depth

Throughput
SIMD width

Conclusion

- Stateless dynamic vectorization is implementable
- Unexpectedly inexpensive
 - ◆ Overhead amortized even for single-issue RISC without FPU
- Scalable
 - ◆ Parallelism in same class as state-of-the-art GPUs
- Minimal software impact
 - ◆ Standard scalar RISC-V instruction set, no proprietary extension
 - ◆ Reuse the RISC-V software infrastructure: gcc and LLVM backends
 - ◆ OS changes to manage ~10K threads?
- One step on the road to single-ISA heterogeneous CPU+GPU

Simty: Generalized SIMT execution on RISC-V

CARRV 2017

Sylvain Collange
INRIA Rennes / IRISA
sylvain.collange@inria.fr

