
GPU architecture part 2: SIMT control flow management

Sylvain Collange
Inria Rennes – Bretagne Atlantique

<http://www.irisa.fr/alf/collange/>
sylvain.collange@inria.fr

ADA - 2017

Outline

- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks
 - ◆ Counters
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

Analogy: Out-of-order microarchitecture

Software

```
void scale(float a, float * X, int n)
{
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}
```

scale:

```
test esi, esi
je .L4
sub esi, 1
xor eax, eax
lea rdx, [4+rsi*4]
.L3:
movss xmm1, DWORD PTR [rdi+rax]
mulss xmm1, xmm0
movss DWORD PTR [rdi+rax], xmm1
add rax, 4
cmp rax, rdx
jne .L3
.L4:
rep
ret
```

Architecture: **sequential** programming model

Hardware

Analogy: Out-of-order microarchitecture

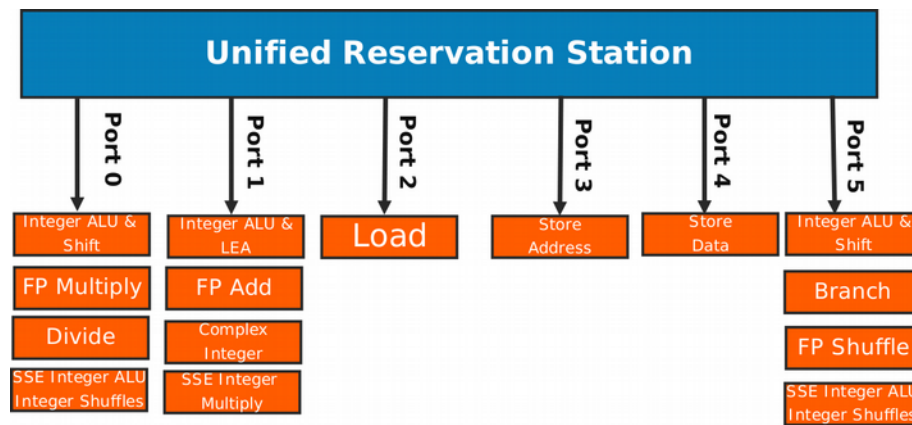
Software

```
void scale(float a, float * X, int n)
{
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}
```

```
scale:
    test    esi, esi
    je      .L4
    sub     esi, 1
    xor     eax, eax
    lea     rdx, [4+rsi*4]
.L3:
    movss   xmm1, DWORD PTR [rdi+rax]
    mulss   xmm1, xmm0
    movss   DWORD PTR [rdi+rax], xmm1
    add     rax, 4
    cmp     rax, rdx
    jne     .L3
.L4:
    rep
    ret
```

Architecture: **sequential** programming model

Hardware datapaths: **dataflow** execution model



Analogy: Out-of-order microarchitecture

Software

```
void scale(float a, float * X, int n)
{
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}
```

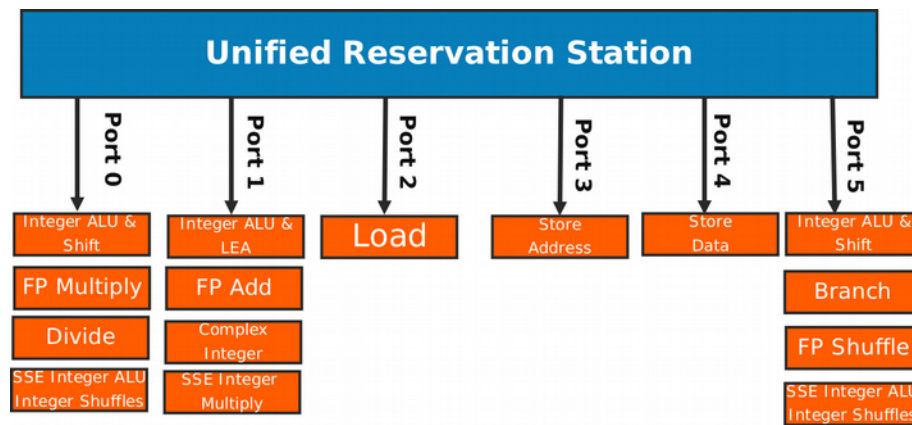
scale:

```
test esi, esi
je .L4
sub esi, 1
xor eax, eax
lea rdx, [4+rsi*4]
.L3:
movss xmm1, DWORD PTR [rdi+rax]
mulss xmm1, xmm0
movss DWORD PTR [rdi+rax], xmm1
add rax, 4
cmp rax, rdx
jne .L3
.L4:
rep
ret
```

Architecture: **sequential** programming model

Dark magic!

Hardware datapaths: **dataflow** execution model



Hardware

Analogy: Out-of-order microarchitecture

Software

```
void scale(float a, float * X, int n)
{
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}
```

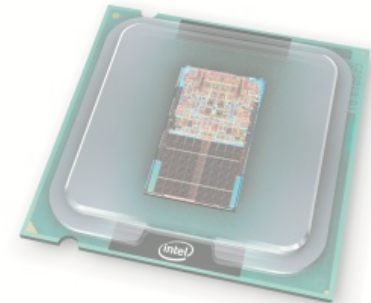
scale:

```
test esi, esi
je .L4
sub esi, 1
xor eax, eax
lea rdx, [4+rsi*4]
.L3:
movss xmm1, DWORD PTR [rdi+rax]
mulss xmm1, xmm0
movss DWORD PTR [rdi+rax], xmm1
add rax, 4
cmp rax, rdx
jne .L3
.L4:
rep
ret
```

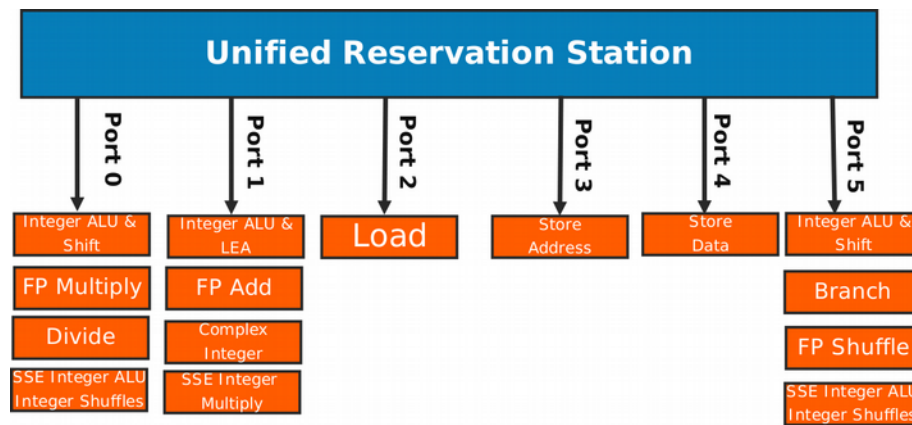
Architecture: **sequential** programming model

Dark magic!

**Out-of-order superscalar
microarchitecture**



Hardware datapaths: **dataflow** execution model

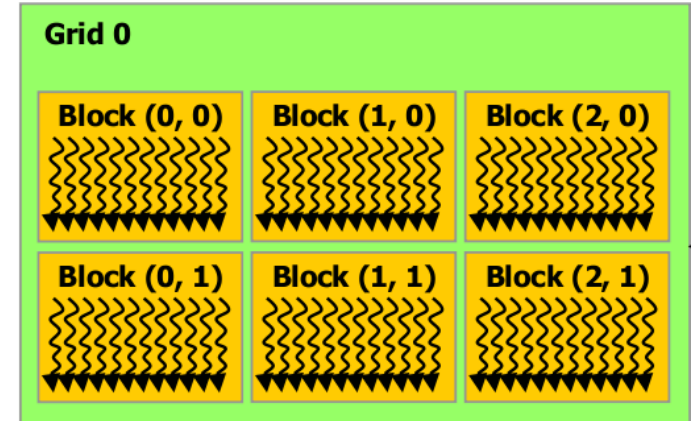


Hardware

GPU microarchitecture: where it fits

Software

```
__global__ void scale(float a, float * X)
{
    unsigned int tid;
    tid = blockIdx.x * blockDim.x
        + threadIdx.x;
    X[tid] = a * X[tid];
}
```

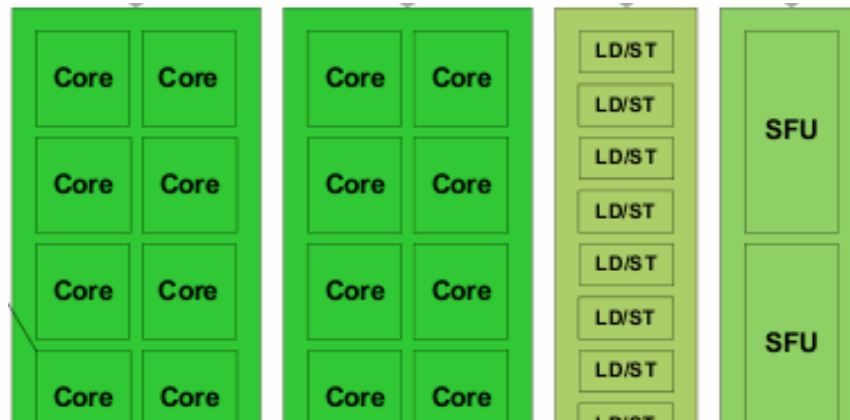
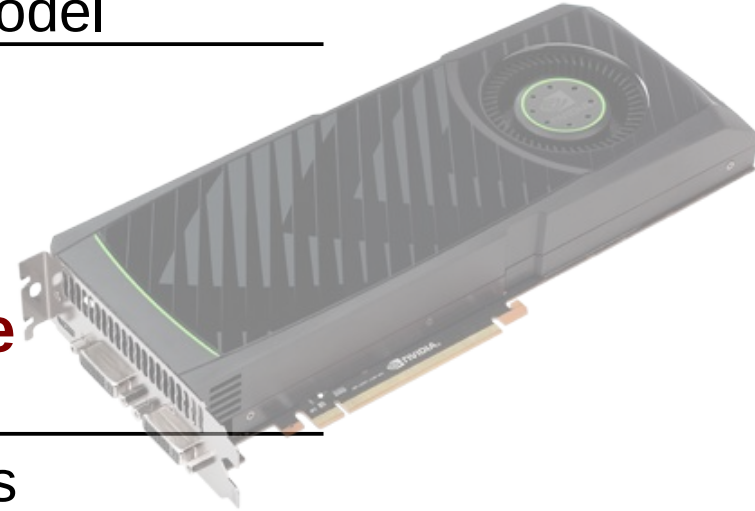


Architecture: **multi-thread** programming model

Dark magic!

SIMT microarchitecture

Hardware datapaths: **SIMD** execution units



Hardware

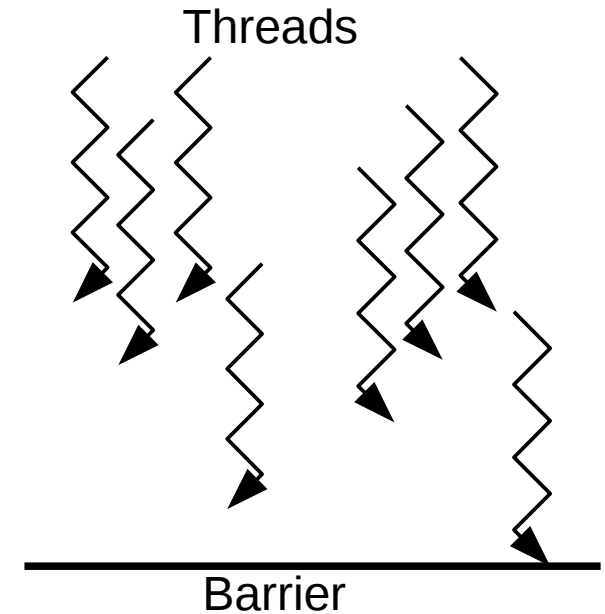
Programmer's view

- Programming model

- ◆ SPMD: Single program, multiple data
- ◆ One *kernel* code, many *threads*
- ◆ Unspecified execution order between explicit synchronization barriers

- Languages

- ◆ Graphics shaders : HLSL, Cg, GLSL
- ◆ GPGPU : C for CUDA, OpenCL



For n threads:
 $X[tid] \leftarrow a * X[tid]$

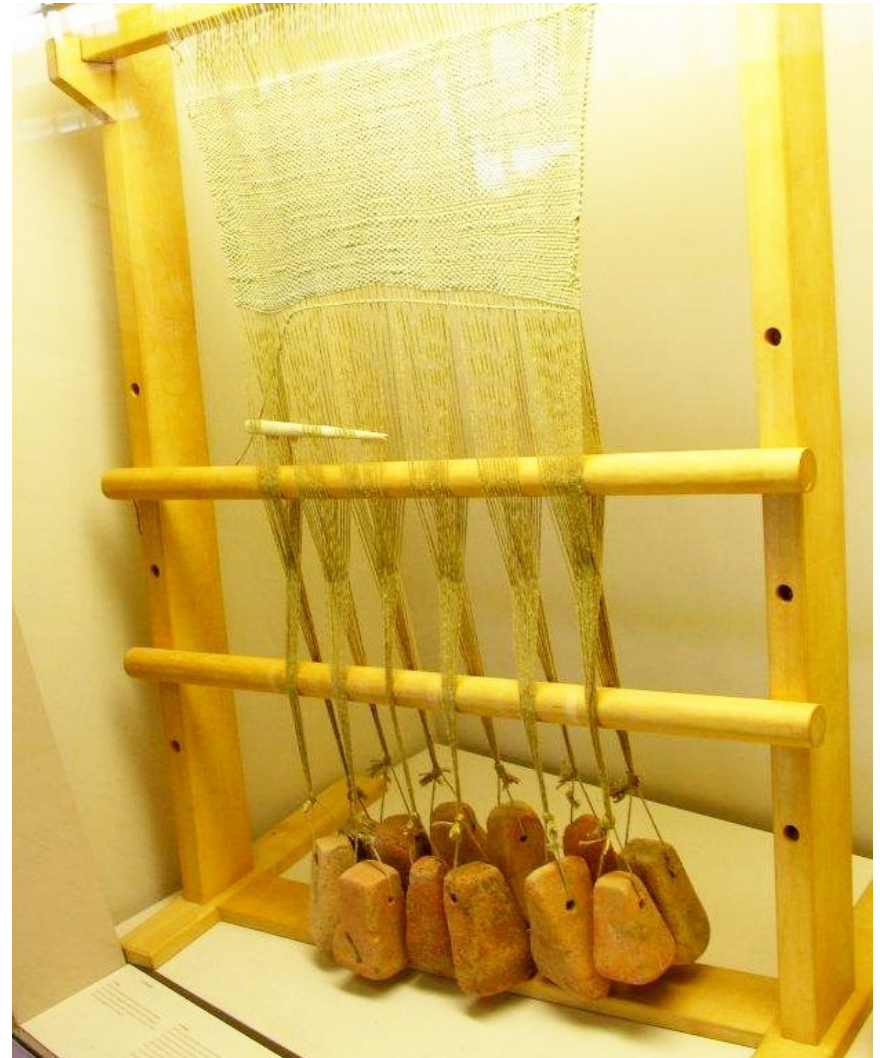
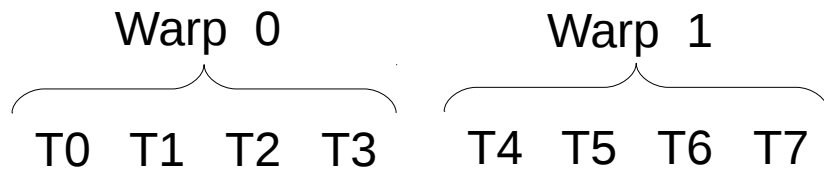
Kernel

Types of control flow

- **Structured** control flow:
single-entry, single exit
properly nested
 - ◆ Conditionals: `if-then-else`
 - ◆ Single-entry, single-exit loops: `while`, `do-while`, `for...`
 - ◆ Function call-return...
- **Unstructured** control flow
 - ◆ `break`, `continue`
 - ◆ `&&` `||` short-circuit evaluation
 - ◆ Exceptions
 - ◆ Coroutines
 - ◆ `goto`, `comefrom`
 - ◆ Everything that is hard to indent

Warps

- Threads are grouped into warps of fixed size



An early SIMT architecture. Musée Gallo-Romain de S^t-Romain-en-Gal, Vienne

Control flow: uniform or divergent

- Control is **uniform** when all threads in the warp follow the same path
- Control is **divergent** when different threads follow different paths

Outcome per thread:

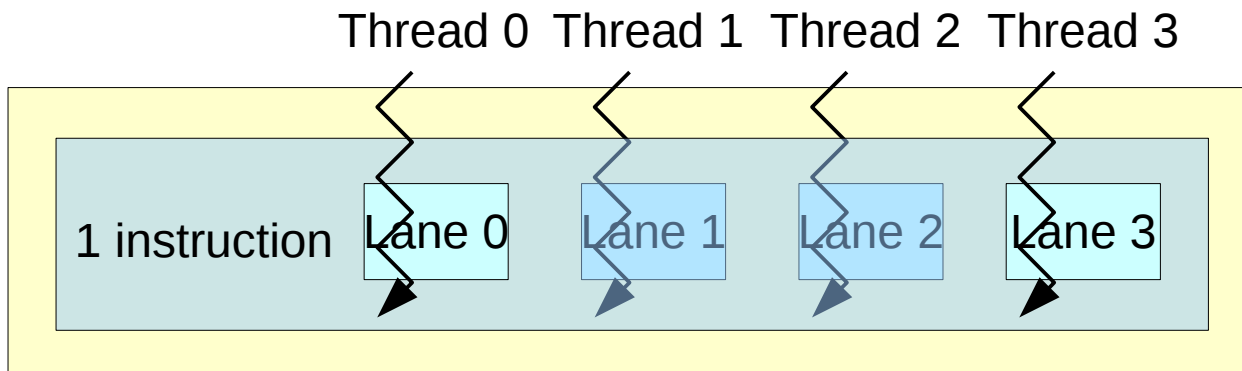
	Warp			
	T0	T1	T2	T3
<code>x = 0;</code>				
<code>// Uniform condition</code>				
<code>if(a[x] > 17) {</code>	1	1	1	1
<code> x = 1;</code>				
<code>}</code>				
<code>// Divergent condition</code>				
<code>if(tid < 2) {</code>	1	1	0	0
<code> x = 2;</code>				
<code>}</code>				

Computer architect view

- SIMD execution inside a warp
 - ◆ One SIMD lane per thread
 - ◆ All SIMD lanes see the same instruction
- Control-flow differentiation using **execution mask**
 - ◆ All instructions controlled with 1 bit per lane
 - 1 → perform instruction
 - 0 → do nothing

Running independent threads in SIMD

- How to keep threads synchronized?
 - ◆ Challenge: divergent control flow
- Rules of the game
 - ◆ One thread per SIMD lane
 - ◆ **Same instruction** on all lanes
 - ◆ Lanes can be individually disabled with **execution mask**
- Which instruction?
- How to compute execution mask?



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

Example: if

Execution mask inside if statement = if condition

		Warp			
		T0	T1	T2	T3
<pre>x = 0; // Uniform condition if(a[x] > 17) { x = 1; }</pre>	if condition:				
	a[x] > 17?	1	1	1	1
	Execution mask:	1	1	1	1
<pre>// Divergent condition if(tid < 2) { x = 2; }</pre>	if condition:				
	tid < 2?	1	1	0	0
	Execution mask:	1	1	0	0

State of the art in 1804

- The Jacquard loom is a GPU



Shaders



Framebuffer

- Multiple warps with per-thread conditional execution
 - ◆ Execution mask given by punched cards
 - ◆ Supports 600 to 800 parallel threads

Conditionals that no thread executes

- Do not waste energy fetching instructions not executed
 - ◆ Skip instructions when execution mask is all-zeroes

		Warp			
		T0	T1	T2	T3
<pre>x = 0; // Uniform condition if(a[x] > 17) { x = 1; } else { x = 0; }</pre>	if condition:				
	a[x] > 17?	1	1	1	1
	Execution mask:	1	1	1	1
<pre>// Divergent condition if(tid < 2) { x = 2; }</pre>	Execution mask:	0	0	0	0

↓ Jump to end-if

- Uniform branches are just usual scalar branches

What about loops?

- Keep looping until all threads exit
 - Mask out threads that have exited the loop

Execution trace:			Warp			
			T0	T1	T2	T3
<pre>i = 0; while(i < tid) { i++; } print(i);</pre>	i = 0;	i=?	0	0	0	0
		i < tid?	0	1	1	1
	i++;	i=?	0	1	1	1
		i < tid?	0	0	1	1
	i++;	i=?	0	1	2	2
		i < tid?	0	0	0	1
	i++;	i=?	0	1	2	3
		i < tid?	0	0	0	0
	No active thread left → restore mask and exit loop					
	print(i);	i=?	0	1	2	3

Time

What about nested control flow?

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

- We need a generic solution!

Outline

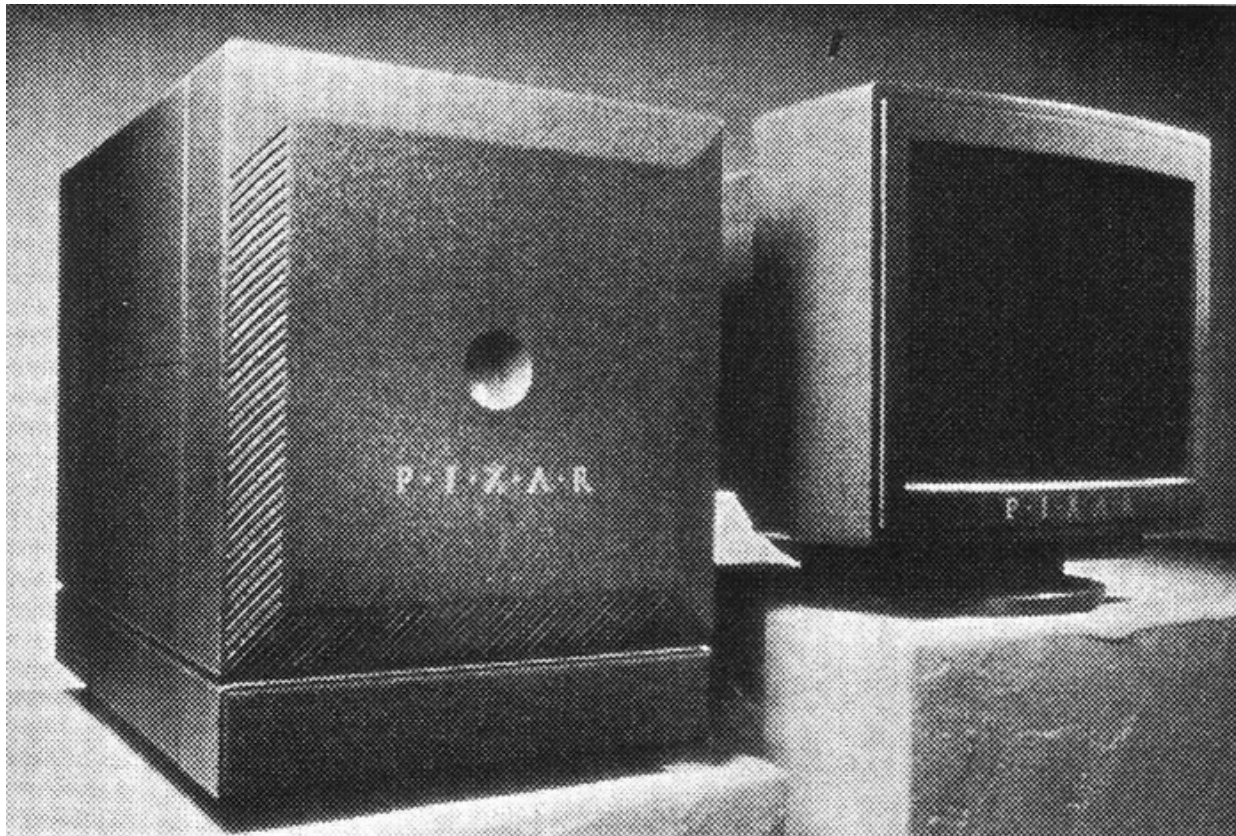
- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks
 - ◆ Counters
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

Quizz

- What do *Star Wars* and GPUs have in common?

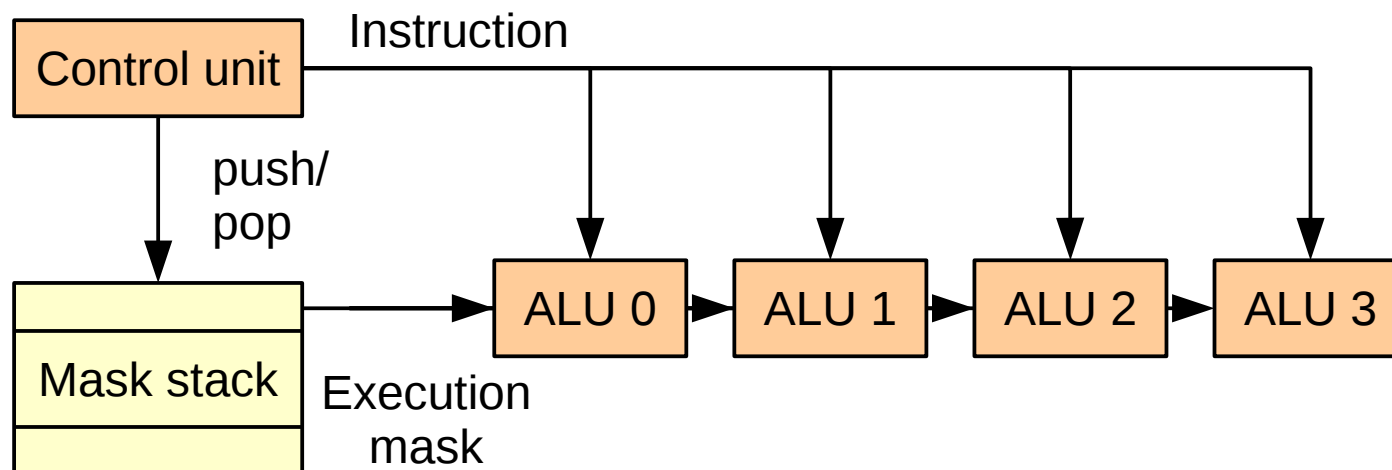
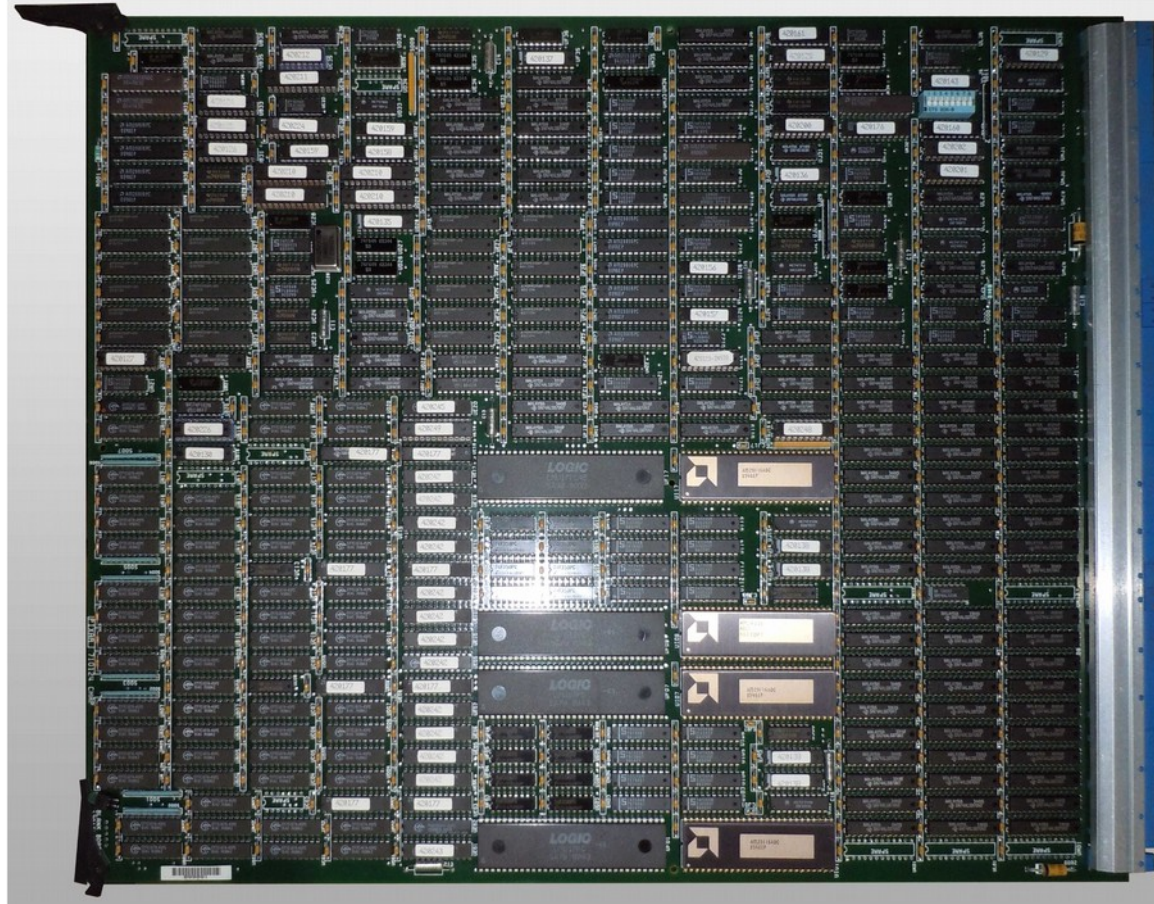
Answer: Pixar!

- In the early 1980's, the *Computer Division* of *Lucasfilm* was designing custom hardware for computer graphics
- Acquired by Steve Jobs in 1986 and became *Pixar*
- Their core product: the *Pixar Image Computer*



- This early GPU handles nested divergent control flow!

Pixar Image Computer: architecture overview



The mask stack of the Pixar Image Computer

Code

x = 0;

```
// Uniform condition
```

```
if(tid > 17) {
```

x = 1;

}

```
// Divergent conditions
```

```
if(tid < 2) {
```

push 

```
if(tid == 0) {
```



push

```
x = 2;
```

```
} pop
```

```
else {
```


push

x = 3;

```

} pop

```

} pop

Mask Stack

1 activity bit / thread

1111

tid=0

tid=2

1111

tid=1

tid=3

1111

1100

1111

1100

1000

1111

1100

1111

1100

0100

1111

1100

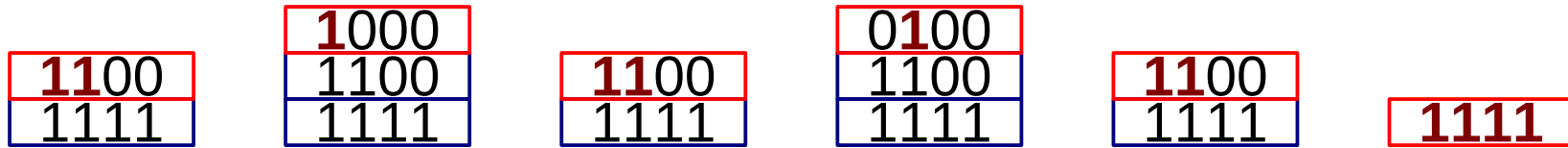
1111

Observation: stack content is a histogram

<table><tr><td>1100</td></tr><tr><td>1111</td></tr></table>	1100	1111	<table><tr><td>1000</td></tr><tr><td>1100</td></tr><tr><td>1111</td></tr></table>	1000	1100	1111	<table><tr><td>1100</td></tr><tr><td>1111</td></tr></table>	1100	1111	<table><tr><td>0100</td></tr><tr><td>1100</td></tr><tr><td>1111</td></tr></table>	0100	1100	1111	<table><tr><td>1100</td></tr><tr><td>1111</td></tr></table>	1100	1111	<table><tr><td>1111</td></tr></table>	1111
1100																		
1111																		
1000																		
1100																		
1111																		
1100																		
1111																		
0100																		
1100																		
1111																		
1100																		
1111																		
1111																		

- On structured control flow: columns of 1s
 - ◆ A thread active at level n is active at all levels $i < n$
 - ◆ Conversely: no “zombie” thread gets revived at level $i > n$ if inactive at n

Observation: stack content is a histogram



- On structured control flow: columns of 1s
 - ◆ A thread active at level n is active at all levels $i < n$
 - ◆ Conversely: no “zombie” thread gets revived at level $i > n$ if inactive at n
- The height of each column of 1s is enough
 - ◆ Alternative implementation: maintain an activity counter for each thread

With activity counters

Code

```
x = 0;
```

```
// Uniform condition
```

```
if(tid > 17) {
```

```
    x = 1;
```

```
}
```

```
// Divergent conditions
```

```
if(tid < 2) {
```

inc

```
    if(tid == 0) {
```

inc

```
        x = 2;
```

dec

```
    }
```

```
    else {
```

inc

```
        x = 3;
```

dec

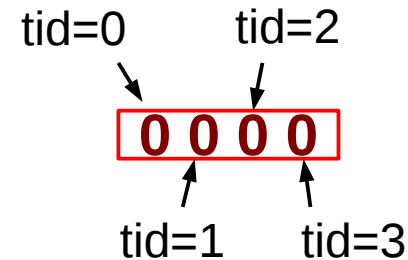
```
    }
```

dec

```
}
```

Counters

1 (in)activity counter / thread



0 0 1 1

0 1 2 2

0 0 1 1

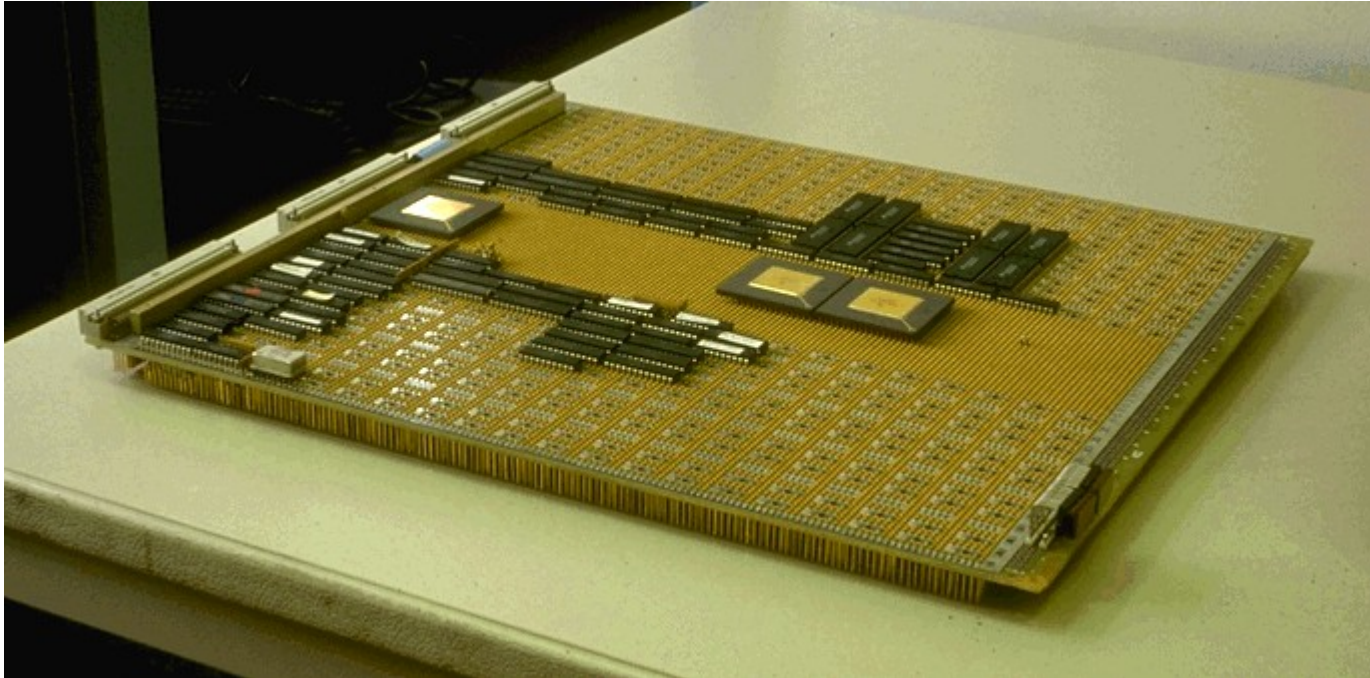
1 0 2 2

0 0 1 1

0 0 0 0

Activity counters in use

- In an SIMD prototype from École des Mines de Paris in 1993



Credits: Ronan Keryell

- In Intel integrated graphics since 2004

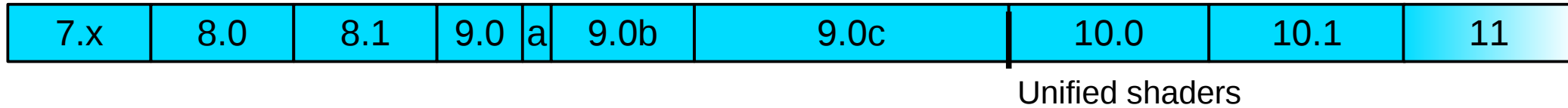


Outline

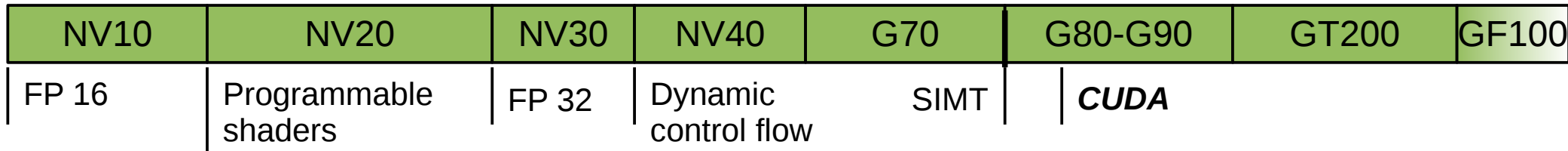
- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks, counters
 - ◆ A compiler perspective
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

Some ancient history

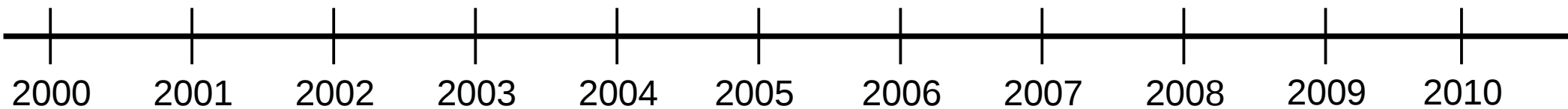
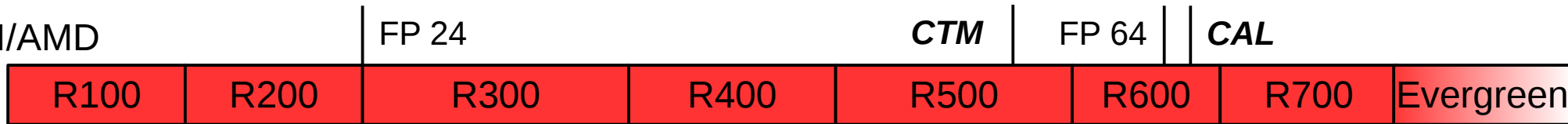
Microsoft DirectX



NVIDIA



ATI/AMD



Early days of programmable shaders

It is 21st century!

- Graphics cards now look and sound like hair dryers



- Graphics shaders are programmed in assembly-like language
 - ◆ Direct3D shader assembly, OpenGL ARB Vertex/Fragment Program...
 - ◆ Control-flow: `if`, `else`, `endif`, `while`, `break`... are assembly instructions
- Graphics driver performs a straightforward translation to GPU-specific machine language

Goto considered harmful?

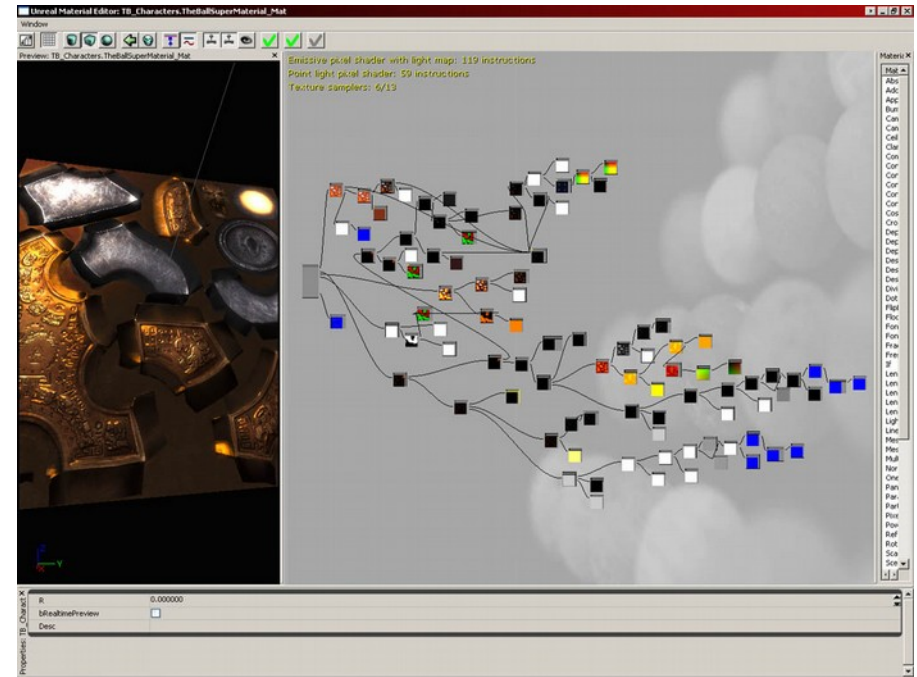
MIPS	NVIDIA Tesla (2007)	NVIDIA Fermi (2010)	Intel GMA Gen4 (2006)	Intel GMA SB (2011)	AMD R500 (2005)	AMD R600 (2007)	AMD Cayman (2011)
j	bar	bar	jmp	jmp	jump	push	push
jal	bra	bpt	if	if	loop	push_else	push_else
jr	brk	bra	iff	else	endloop	pop	pop
syscall	brkpt	brk	else	endif	rep	loop_start	push_wqm
	cal	brx	endif	case	endrep	loop_start_no_al	pop_wqm
	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	mrest	return		else	loop_start_no_al
	trap	pbk	push	fork		call	loop_start_dx10
	.s	pret	pop			call_fs	loop_end
		ret				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 instructions in some CPU
and GPU instruction sets

- GPUs: instruction set expresses control flow structure
 - Where should we stop?

Next: compilers for GPU code

- High-level shader languages
 - ◆ C-like: HLSL, GLSL, Cg
 - ◆ Then visual languages (UDK)
- General-purpose languages
 - ◆ CUDA, OpenCL
 - ◆ Then directive-based: OpenACC, OpenMP 4
 - ◆ Python (Numba)...
- Incorporate function calls, switch-case, && and ||...
- Demands a compiler infrastructure
 - ◆ A Just-In-Time compiler in graphics drivers



A typical GPU compiler

- First: turns all structured control flow into gotos, generates intermediate representation (IR)
 - ◆ e.g. Nvidia PTX, Khronos SPIR, llvm IR
- Then: performs various compiler optimizations on IR
- Finally: reconstructs structured control flow back from gotos to emit machine code
 - ◆ Not necessarily the same as the original source!

Issues of stack-based implementations

If GPU threads are actual threads, they can synchronize?

- e.g. using semaphores, mutexes, condition variables...
- Problem: SIMT-induced livelock

```
while(!acquire(lock)) {}  
...  
release(lock)
```

Example: critical section

Thread 0 acquires the lock,

keeps looping with other threads of the warp waiting for the lock.

Infinite loop, lock never released.

- Stack-based SIMT divergence control can cause starvation!

Issues of stack-based implementations

- Are all control flow optimizations valid in SIMT?
- What about context switches?
 - ◆ e.g. migrate one single thread of a warp
 - ◆ Challenging to do with a stack
- Truly general-purpose computing demands more flexible techniques

Outline

- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks, counters
 - ◆ A compiler perspective
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

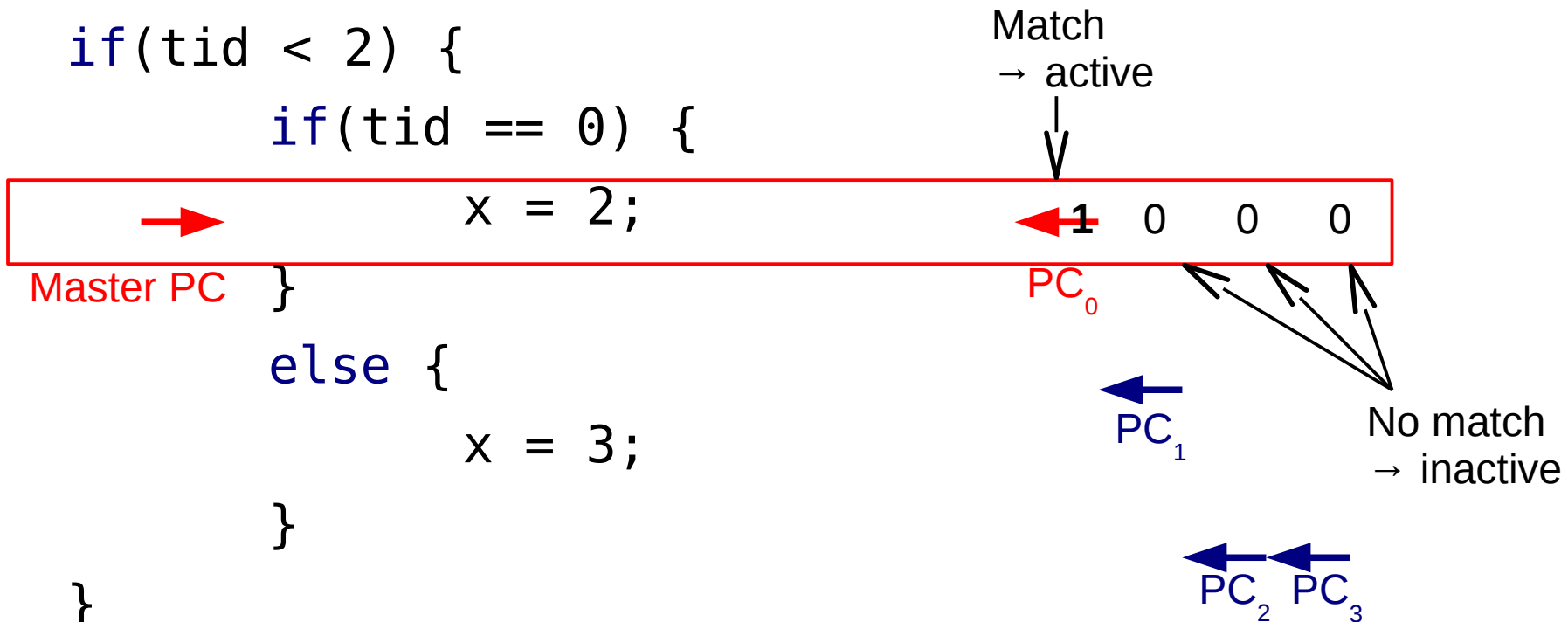
With 1 PC / thread

Code

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

Program Counters (PCs)

tid= 0 1 2 3

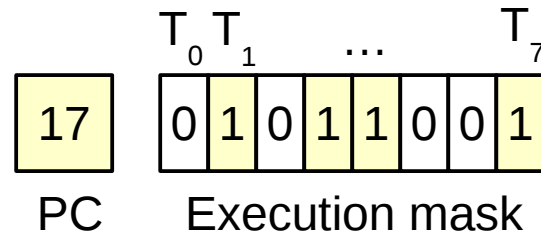


Mask stacks vs. per-thread PCs

- Before: stack, counters
 - ◆ $O(n)$, **$O(\log n)$** memory
 n = nesting depth
 - ◆ **1 R/W port** to memory
 - ◆ **Exceptions**: stack overflow, underflow
- Vector semantics
 - ◆ Structured control flow only
 - ◆ Specific instruction sets
- After: multiple PCs
 - ◆ **$O(1)$** memory
 - ◆ **No shared state**
 - ◆ Allows thread **suspension, restart, migration**
- Multi-thread semantics
 - ◆ Traditional languages, compilers
 - ◆ **Traditional instruction sets**
- **Can be mixed with MIMD**
- **Straightforward implementation is more expensive**

Path-based control flow tracking

- A **path** is characterized by a PC and execution mask



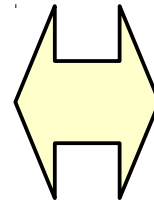
- The mask encodes the **set of threads** that have this PC

$\{T_1, T_3, T_4, T_7\}$ have PC 17

A list of paths represents a vector of PCs

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

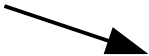
List of paths

- Worst case: 1 path per thread
 - ◆ Path list size is bounded
- PC vector and path list are **equivalent**
 - ◆ You can switch freely between MIMD thinking and SIMD thinking!

Pipeline overview

- Select an active path

Active

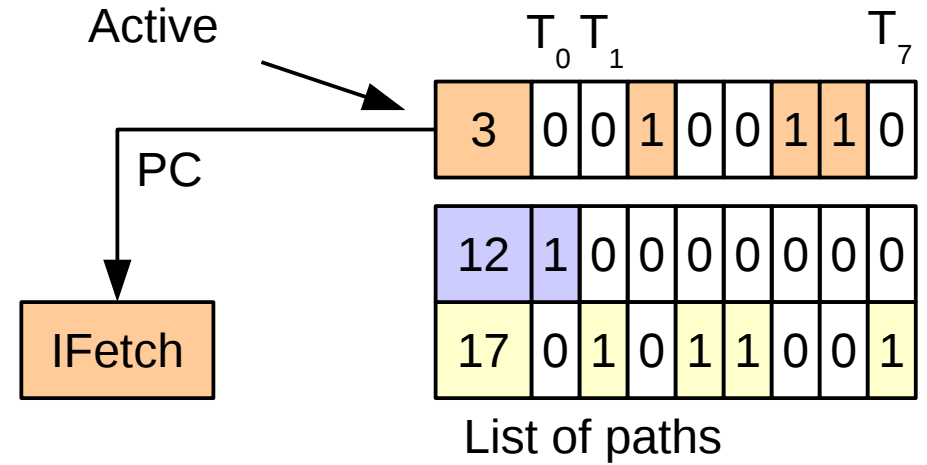


	T_0	T_1						T_7
3	0	0	1	0	0	1	1	0
12	1	0	0	0	0	0	0	0
17	0	1	0	1	1	0	0	1

List of paths

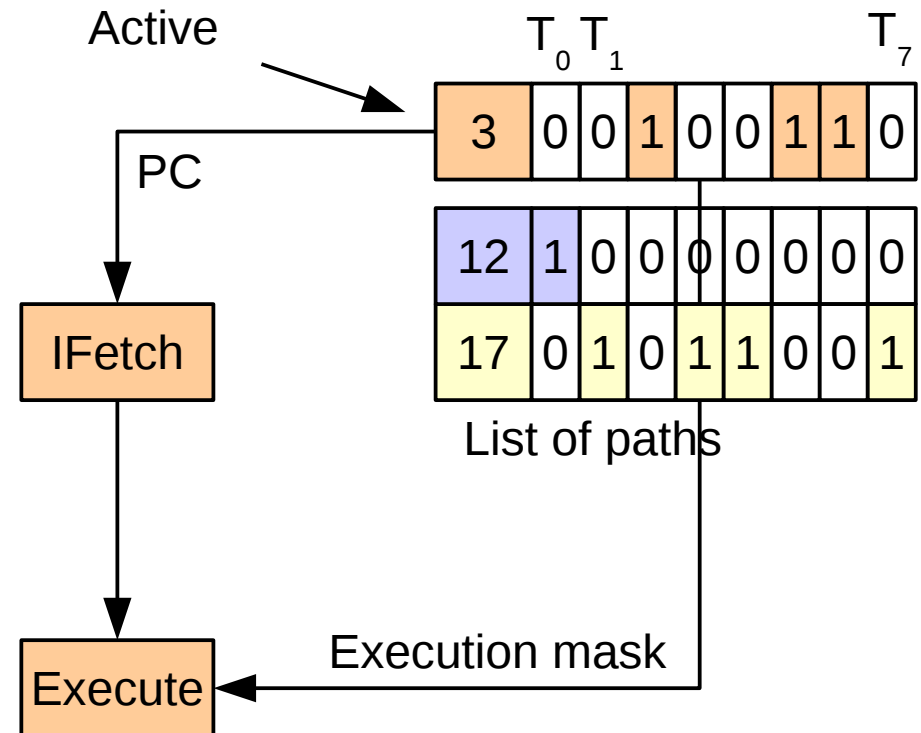
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path



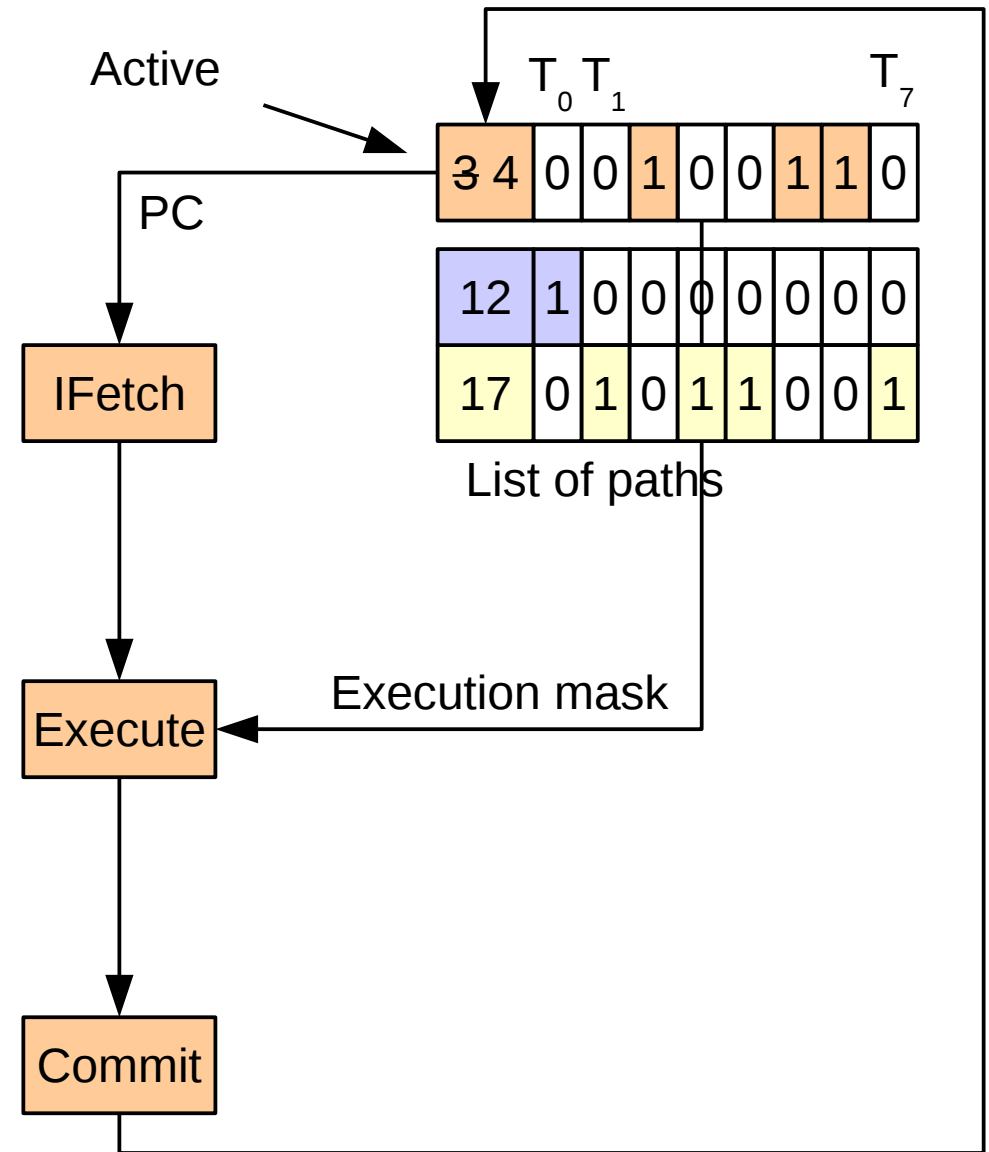
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path
- Execute with execution mask of active path



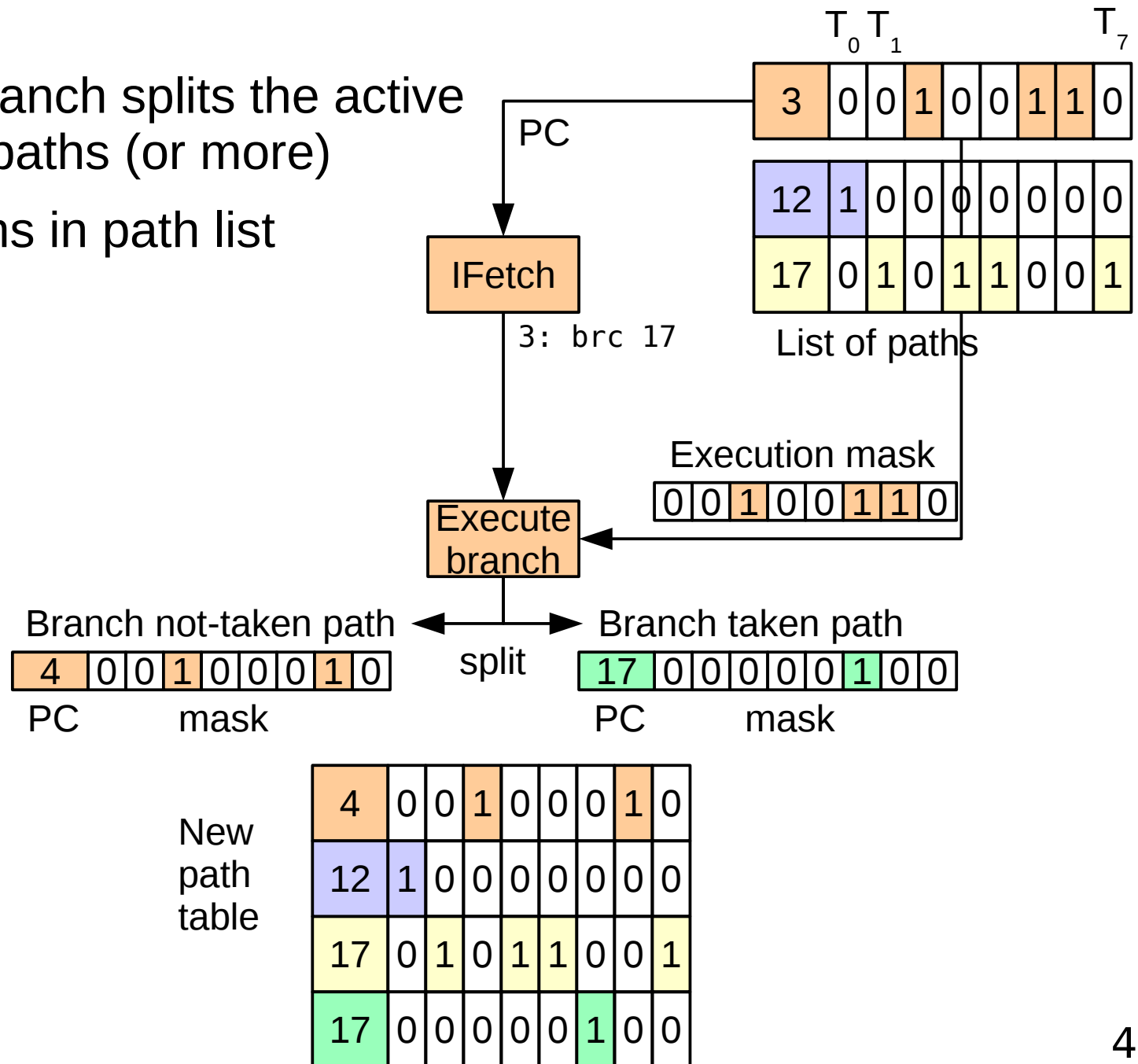
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path
- Execute with execution mask of active path
- For uniform instruction: update PC of active mask



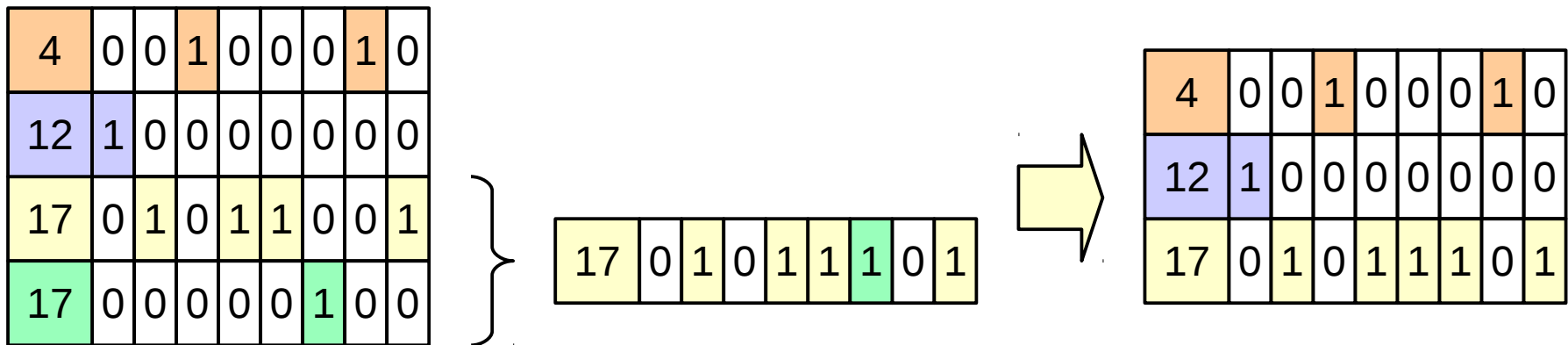
Divergent branch is path insertion

- A divergent branch splits the active path into two paths (or more)
- Insert the paths in path list



Convergence is path fusion

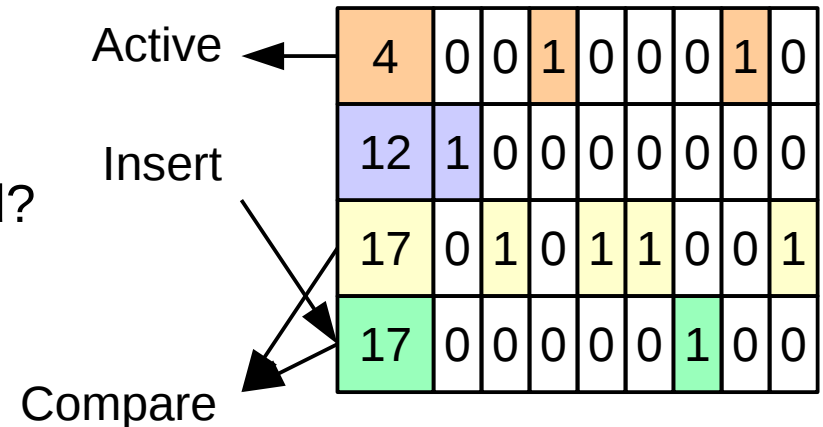
- When two paths have the same PC, we can merge them
 - ◆ New set of threads is the union of former sets
 - ◆ New execution mask is bitwise OR of former masks



Path scheduling is graph traversal

- Degrees of freedom

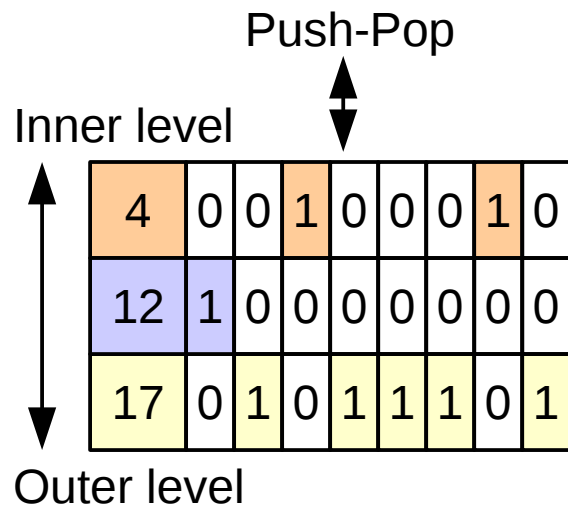
- Which path is the active path?
- At which place are new paths inserted?
- When and where do we check for convergence?



- Different answers yield different policies

Depth-first graph traversal

- Remember graph algorithm theory
 - ◆ Depth-first graph traversal using a stack worklist
- Path list as a stack = depth-first traversal of the control-flow graph
 - ◆ Most deeply nested levels first



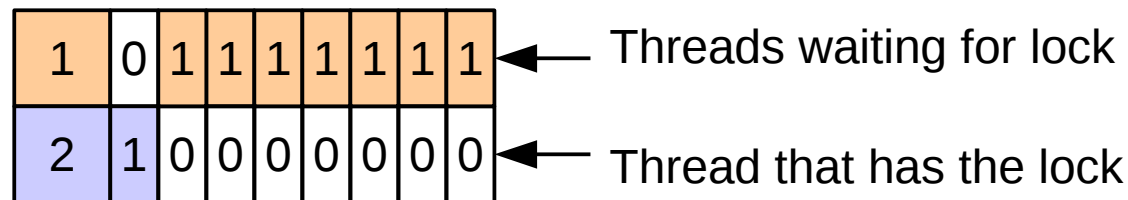
```
x = 0;  
// Uniform condition  
if(tid > 17) {  
    x = 1;  
}  
// Divergent conditions  
if(tid < 2) {  
    if(tid == 0) {  
        x = 2;  
    }  
    else {  
        x = 3;  
    }  
}
```

Question: is this the same as Pixar-style mask stack? Why?

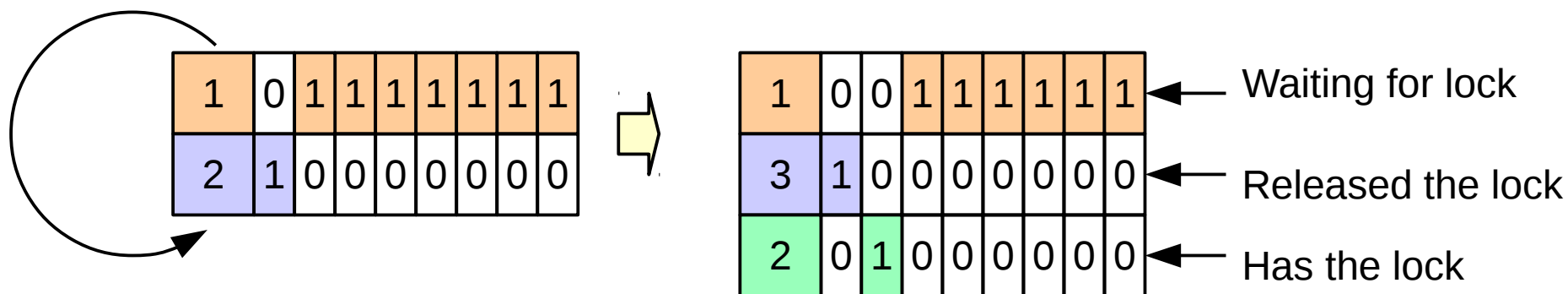
Breadth-first graph traversal

- Goal: guarantee forward progress to avoid SIMT-induced livelocks

```
while(!acquire(lock)) {  
1:  }  
2:...  
   release(lock)  
3:
```



- Path list as a queue: follow paths in round-robin



- Drawback: may delay convergence

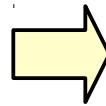
Limitations of static scheduling orders

- Stack works well for structured control flow
 - ◆ Convergence happens in reverse order of divergence
- But not so much for unstructured control flow
 - ◆ Divergence and convergence order do not match

Priority-based graph traversal

- Sort the path list based on its contents

17	0	1	0	1	1	0	0	1
3	0	0	1	0	0	1	1	0
12	1	0	0	0	0	0	0	0



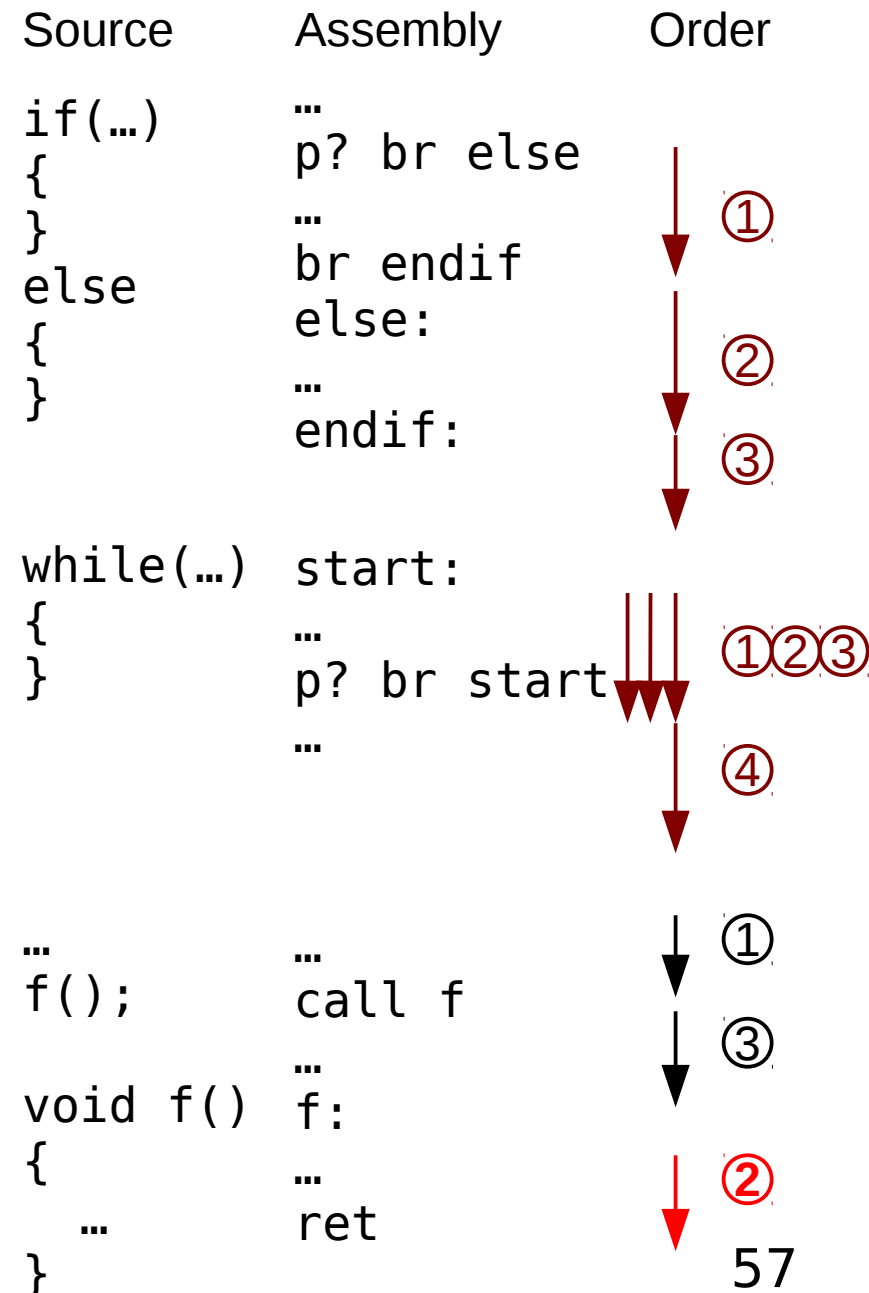
3	0	0	1	0	0	1	1	0
12	1	0	0	0	0	0	0	0
17	0	1	0	1	1	0	0	1

- Ordering paths by priority enables a **scheduling policy**

Scheduling policy: min(SP:PC)

Which PC to choose as master PC ?

- Conditionals, loops
 - ◆ Order of code addresses
 - ◆ min(PC)
- Functions
 - ◆ Favor max nesting depth
 - ◆ min(SP)
- With compiler support
 - ◆ Unstructured control flow too
 - ◆ No code duplication
 - ◆ Full backward and forward compatibility



Convergence with min(PC)-based policies

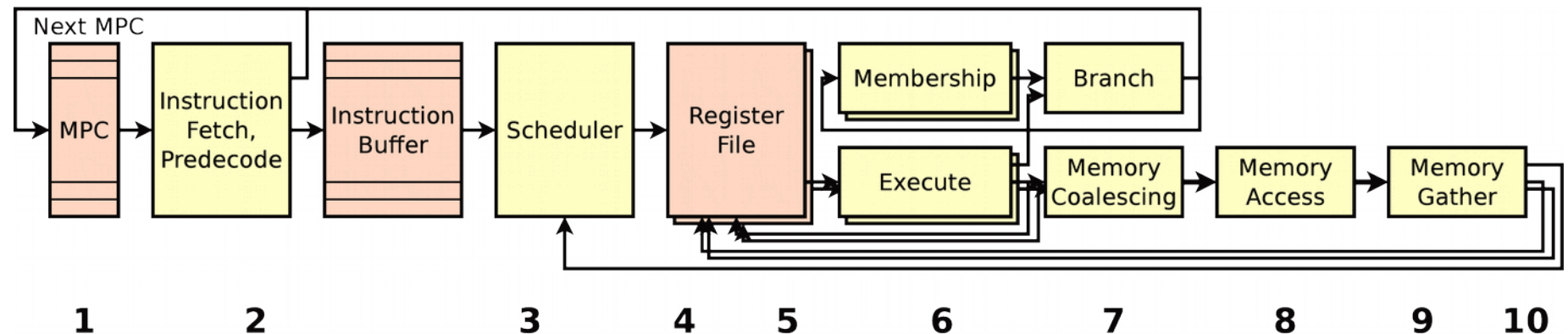
- Sorting by PC groups paths of equal PC together

4	0	0	1	0	0	0	1	0
12	1	0	0	0	0	0	0	0
17	0	1	0	1	1	0	0	1
17	0	0	0	0	0	1	0	0

- Priority order: active path is top entry
- Convergence detection:
only needed between top entry and following entries
 - ◆ No need for associative lookup

Advertisement: Simty, a SIMT CPU

- Proof of concept for priority-based SIMT
 - ◆ Written in synthesizable VHDL
 - ◆ Runs the RISC-V instruction set (RV32I)
 - ◆ Fully parametrizable warp size, warp count
 - ◆ 10-stage pipeline



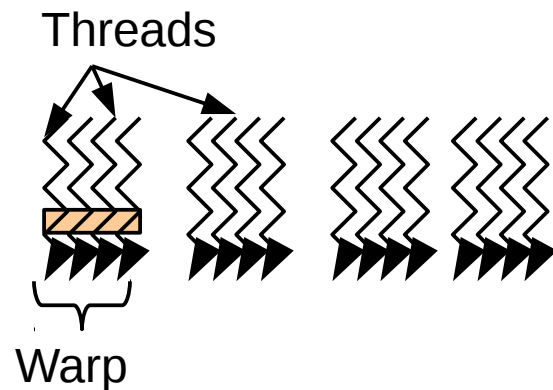
Outline

- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks, counters
 - ◆ A compiler perspective
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

SIMT vs. multi-core + explicit SIMD

- SIMT

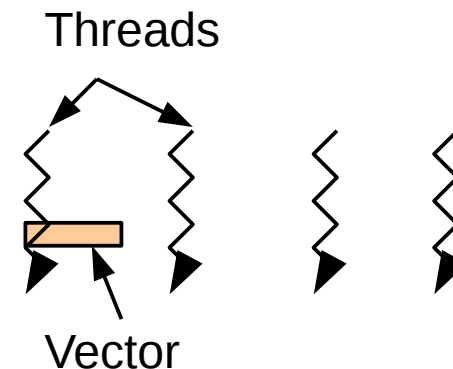
- ◆ All parallelism expressed using threads
- ◆ Warp size implementation-defined
- ◆ Dynamic vectorization



Example: Nvidia GPUs

- Multi-core + explicit SIMD

- ◆ Combination of threads, vectors
- ◆ Vector length fixed at compile-time
- ◆ Static vectorization



Example: most CPUs, Intel Xeon Phi, AMD GCN GPUs

- Are these models equivalent?

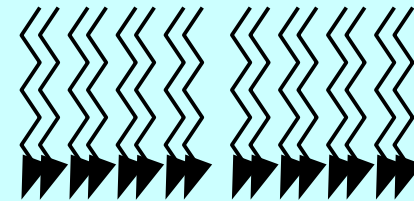
Bridging the gap between SPMD and SIMD

Software: *OpenMP, graphics shaders, OpenCL, CUDA...*

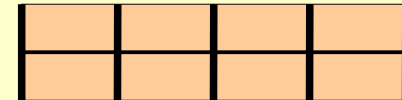
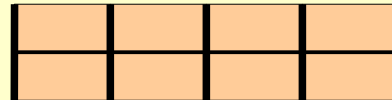
1 kernel

```
kernel void scale(float a, float * X) {  
    X[tid] = a * X[tid];  
}
```

Many threads



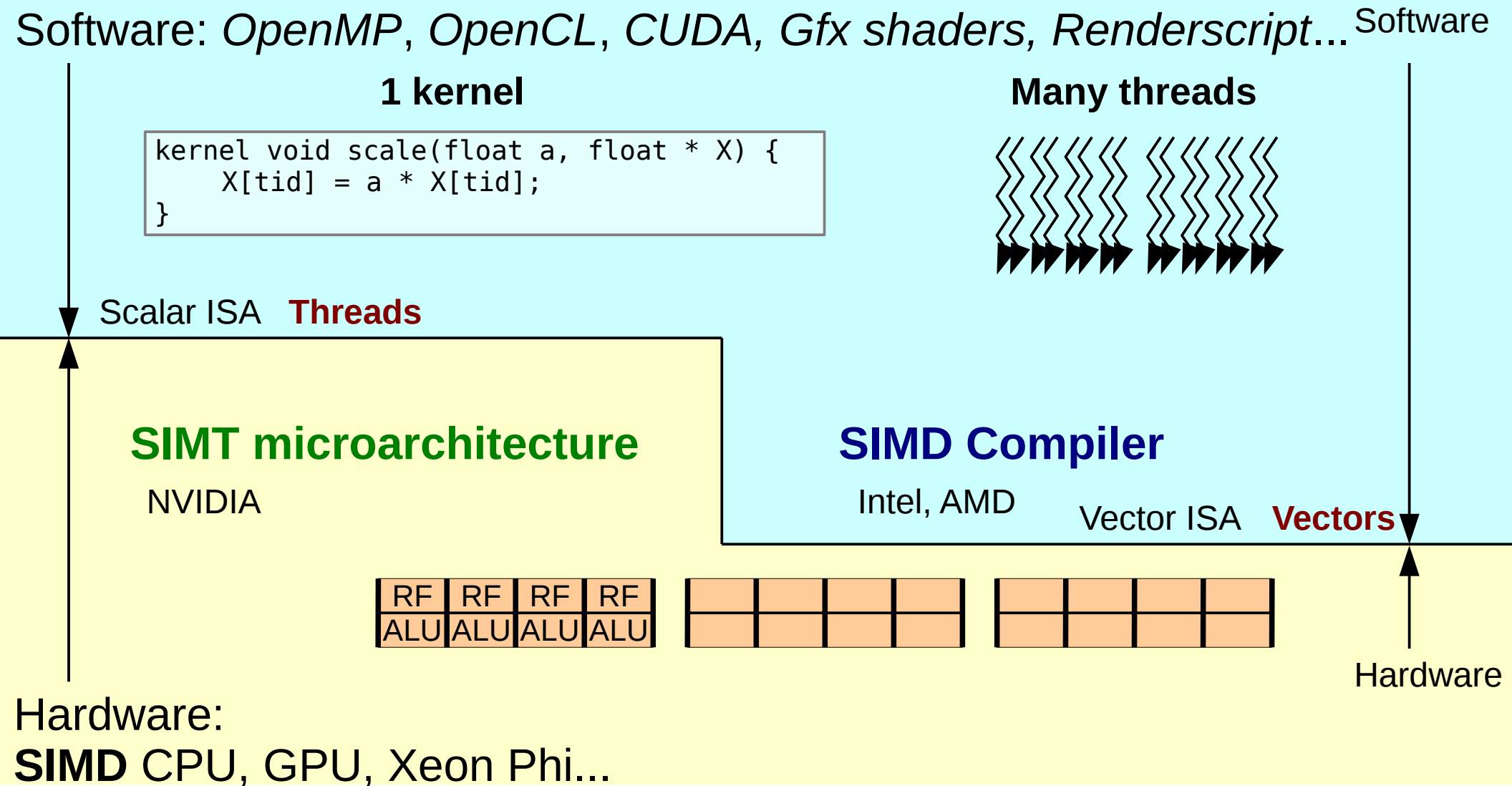
Software



Hardware

Hardware:
SIMD CPU, GPU, Xeon Phi...

SPMD to SIMD: hardware or software ?



- Which is best? : open question
- Combine both approaches?

Tracking control flow in software

- Use cases
 - ◆ Compiling shaders and OpenCL for AMD GCN GPUs
 - ◆ Compiling OpenCL for Xeon Phi
 - ◆ ispc: Intel SPMD Program Compiler, targets various SIMD instruction sets
- Compiler generates code to compute execution masks and branch directions
 - ◆ Same techniques as hardware-based SIMT
 - ◆ But different set of possible optimization

Compiling SPMD to predicated SIMD

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

Compiling SPMD to predicated SIMD

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

```
(m0) mov x←0      // m0 is current mask
(m0) cmp c←tid>17  // vector comparison
      and m1←m0&c  // compute if mask
      jcc(m1=0) endif1 // skip if null
(m1) mov x←1
endif1:
(m0) cmp c←tid<2
      and m2←m0&c
      jcc(m2=0) endif2
(m2) cmp c←tid==0
      and m3←m2&c
      jcc(m3=0) else
(m3) mov x←2
else:
      and m4←m2&~c
      jcc(m4=0) endif2
(m4) mov x←3
endif2:
```

Benefits and shortcomings of s/w SIMT

Benefits

- No stack structure to maintain
 - ◆ Use mask registers directly
 - ◆ Register allocation takes care of reuse and spills to memory
- Compiler knowing precise execution order enables more optimizations
 - ◆ Turn masking into “zeroing”: critical for out-of-order architectures
 - ◆ *Scalarization*: demoting uniform vectors into scalars

Shortcomings

- Every branch is divergent unless proven otherwise
 - ◆ Need to allocate mask register either way
- Restricts freedom of microarchitecture for runtime optimization

Scalars in SPMD code

- Some values and operations are inherently scalar
- ◆ Loop counters, addresses of consecutive accesses...
- ◆ Same value values for all threads of a warp
Uniform vector
- ◆ Or sequence of evenly-spaced values
Affine vector

SPMD code

```

mov    i ← tid
loop:
load   t ← X[i]
mul    t ← axt
store  X[i] ← t
add    i ← i+tnum
branch i<n? loop

```

Thread

0 1 2 3 ...

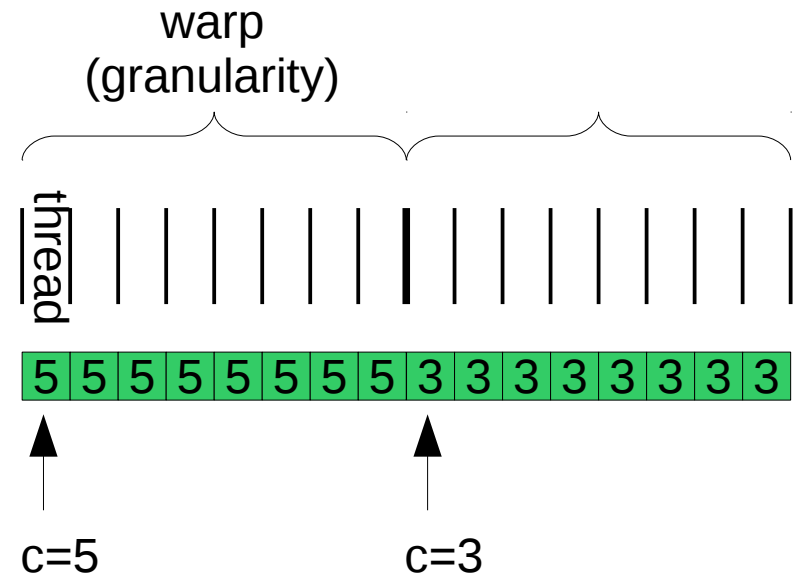
load
mul
store
add
bran

[illegible][illegible]

Uniform and affine vectors

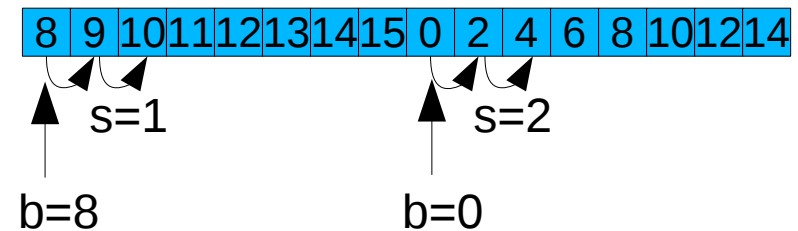
- **Uniform** vector

- ◆ In a warp, $v[i] = c$
- ◆ Value does not depend on lane ID



- **Affine** vector

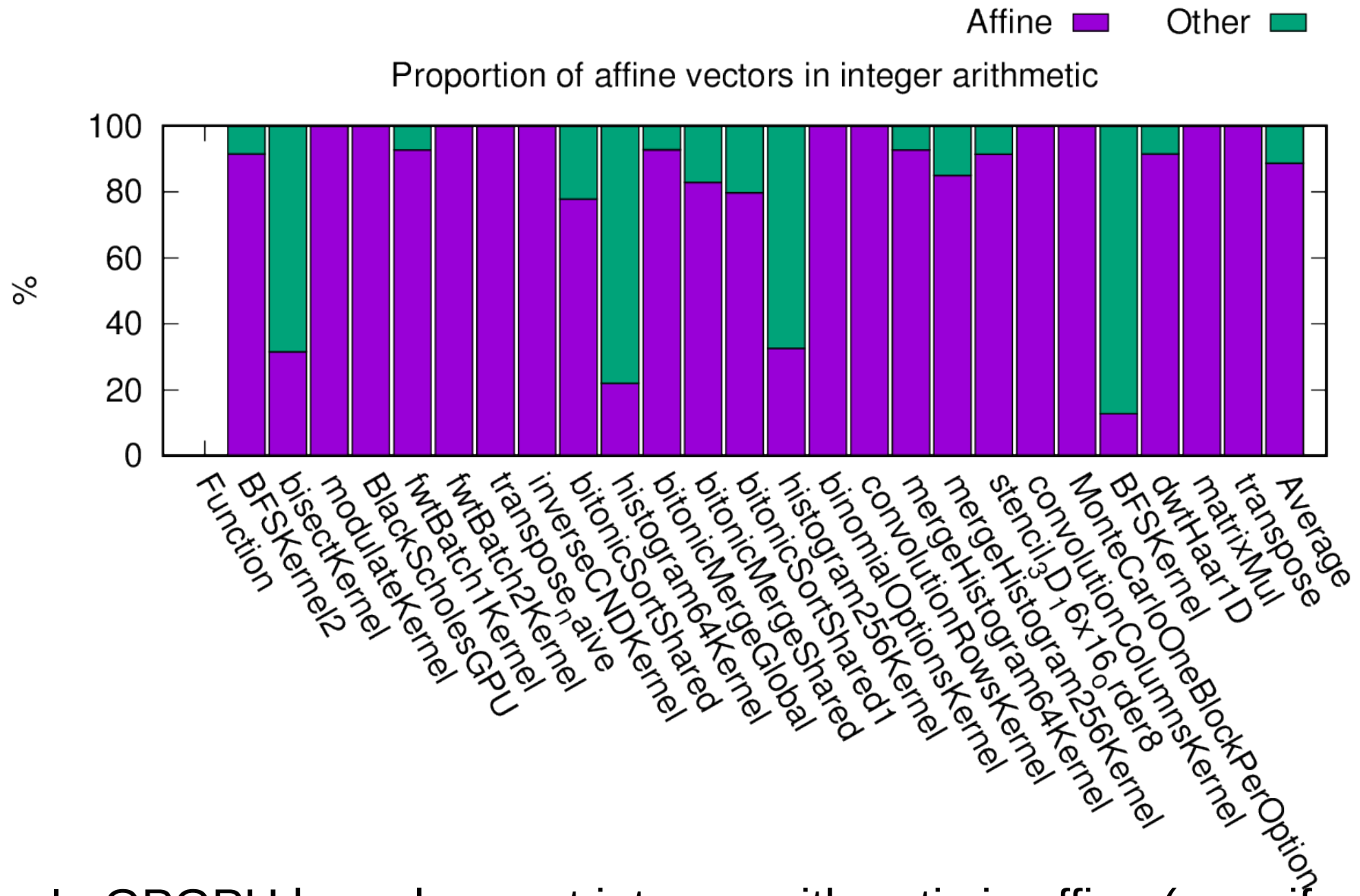
- ◆ In a warp, $v[i] = b + i s$
- ◆ Base b , stride s
- ◆ Affine relation between value and lane ID



- **Generic** vector : anything else



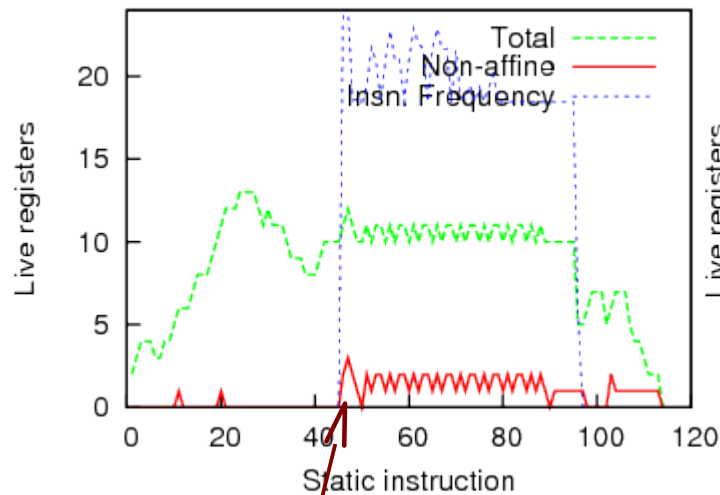
Shocking truth: most vectors are scalars in disguise



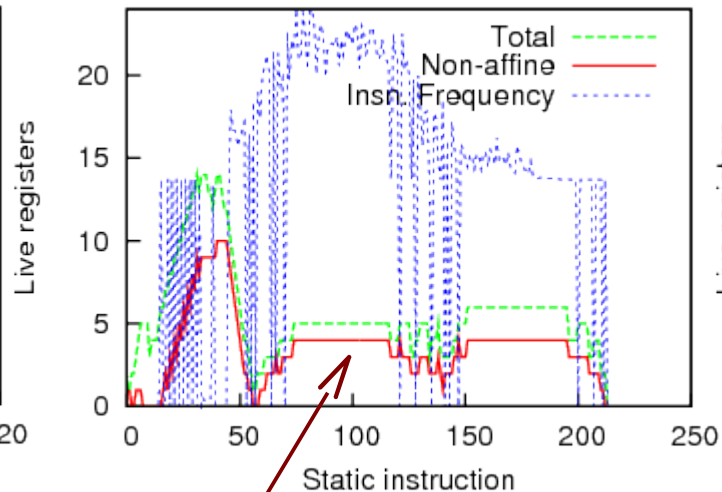
- In GPGPU kernels, most integer arithmetic is affine (or uniform)
 - ◆ i.e. not floating point, not graphics shaders

What is inside a GPU register file?

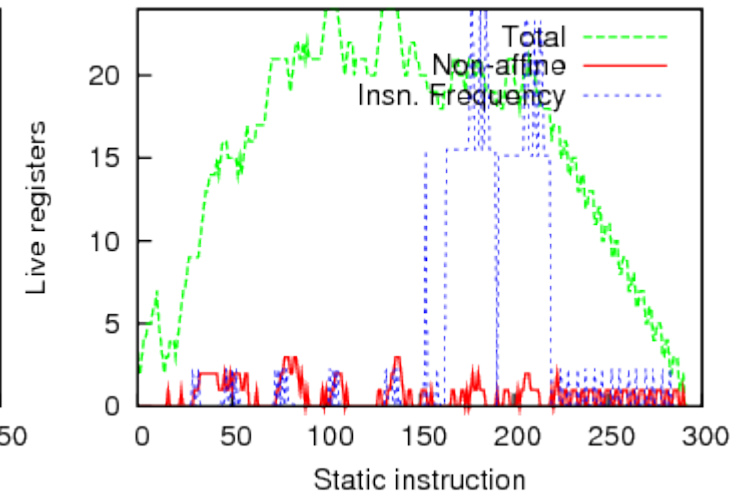
- Non-affine registers alive in inner loop:



MatrixMul: **3** non-affine / **14**



Convolution: **4** non-affine in
hotspot / **14**



Needleman-Wunsch:
2 non-affine / **24**

- 50% - 92% of GPU RF contains affine variables
 - More than register reads: non-affine variables are short-lived
 - Very high potential for register pressure reduction in GPGPU apps

Scalarization

- Explicit SIMD architectures have scalar units
 - ◆ Intel Xeon Phi: has good old x86
 - ◆ AMD GCN GPUs: have scalar units and registers
- Scalarization optimization demotes uniform and affine vectors into scalars
 - ◆ Vector instructions → scalar instructions
 - ◆ Vector registers → scalar registers
 - ◆ SIMT branches → uniform (scalar) branches
 - ◆ Gather-scatter load-store → vector load-store or broadcast
- *Divergence analysis* guides scalarization
 - ◆ Compiler magic not explained here

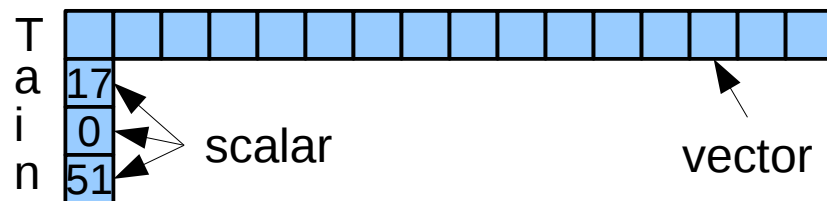
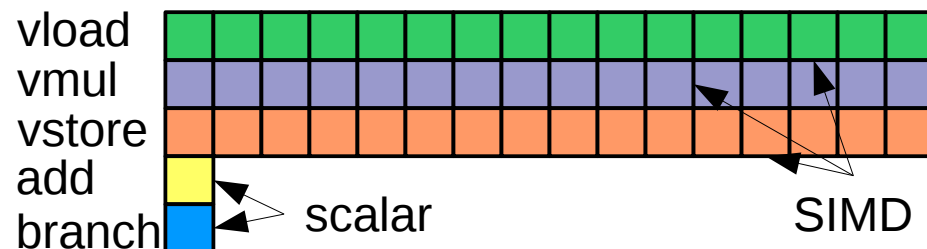
After scalarization

- Obvious benefits
 - ◆ Scalar registers instead of vector
 - ◆ Scalar instructions instead of vector
- Less obvious benefits
 - ◆ Contiguous vector load, store
 - ◆ Scalar branches, no masking
 - ◆ Affine vector \rightarrow single scalar: stride has been constant-propagated!
 - ◆ No dependency between scalar and vector code except through loads and stores: enables decoupling

SIMD+scalar code

```
mov    i ← 0
loop:
  vload  T ← X[i]
  vmul   T ← a×T
  vstore X[i] ← T
  add    i ← i+16
  branch i<n? loop
```

- Instructions



Scalarization across function calls

Which parameters are uniform – affine?

```
kernel void scale(float a, float * X)
{
    // Called for each thread tid
    X[tid] = mul(a, X[tid]);
}
```

```
float mul(float u, float v)
{
    return u * v;
}
```

```
kernel void scale2(float a, float * X)
{
    // Called for each thread tid
    mul_ptr(&a, &X[tid]);
}
```

```
void mul_ptr(float* u, float *v)
{
    *v = (*u) * (*v);
}
```

- Depends on call site

- ◆ Not visible to compiler before link-time,
or requires interprocedural optimization (expensive)
- ◆ Different call sites may have different set of uniform/affine parameters

Typing-based approach

Used in Intel Cilk+

- Programmer qualifies parameters explicitly

```
--declspec (vector uniform(u)) float mul(float u, float v)
{
    return u * v;
}
```

```
--declspec (vector uniform(u) linear(v)) void mul_ptr(float* u, float *v)
{
    *v = (*u) * (*v);
}
```

- Different variations are C++ function overloads
- By default, everything is a generic vector

No automatic solution!

Scalarization with hardware-based SIMT?

- Your thoughts?

Outline

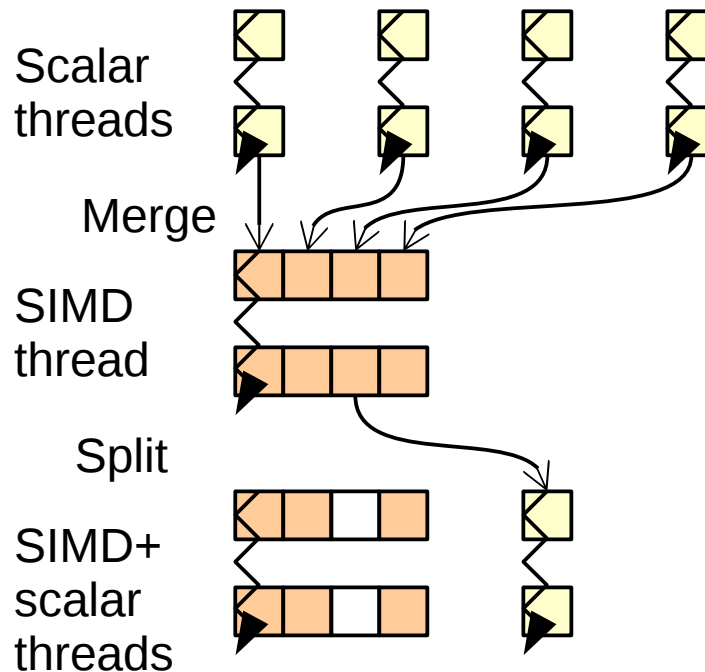
- Running SPMD software on SIMD hardware
 - ◆ Context: software and hardware
 - ◆ The control flow divergence problem
- Stack-based control flow tracking
 - ◆ Stacks, counters
 - ◆ A compiler perspective
- Path-based control flow tracking
 - ◆ The idea: use PCs
 - ◆ Implementation: path list
 - ◆ Applications
- Software approaches
 - ◆ Use cases and principle
 - ◆ Scalarization
- Research directions

Challenge: out of order SIMT

- Has long considered unfeasible for low-power cores
- Empirical evidence show that it is feasible
 - ◆ Most low-power ARM application processors are out-of-order
ARM Cortex A9, A12, A15, A57, Qualcomm Krait
<5W power envelope
 - ◆ Next Intel *Xeon Phi (Knights Landing)* is OoO
70+ OoO cores with 512-bit SIMD units on a chip
- Overhead of OoO amortized by wide SIMD units
 - ◆ Cost of control does not depend on vector length
- Need to adapt OoO to SIMT execution
 - ◆ Main challenges: branch prediction and register renaming

Challenge: improved static vectorization?

- Software equivalent to path traversal is still unknown
- Can we use compiler techniques to achieve SIMT flexibility on existing explicit SIMD architectures?
 - ◆ e.g. Merge scalar threads into a SIMD thread at barriers, split SIMD thread into scalar threads when control flow may diverge



- ◆ Then add scalarization to the mix