
GPU microarchitecture: Revisiting the SIMT execution model

January 2021

Caroline Collange (she/her)

Inria Rennes – Bretagne Atlantique
<https://team.inria.fr/pacap/members/collange/>
caroline.collange@inria.fr



Outline

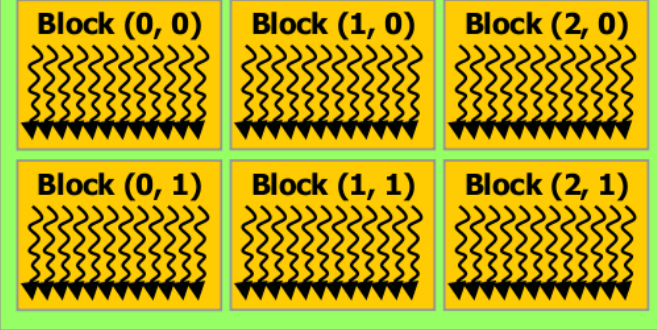
- Running SPMD software on SIMD hardware
 - Context: software and hardware
 - The control flow divergence problem
- Stack-based control flow tracking
 - Stacks
 - Counters
- Path-based control flow tracking
 - The idea: use PCs
 - Implementation: path list
 - Applications

What is GPU microarchitecture?

Software

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

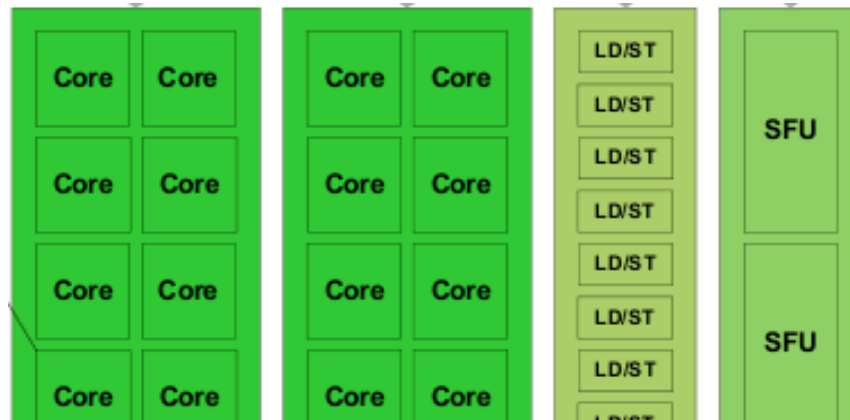
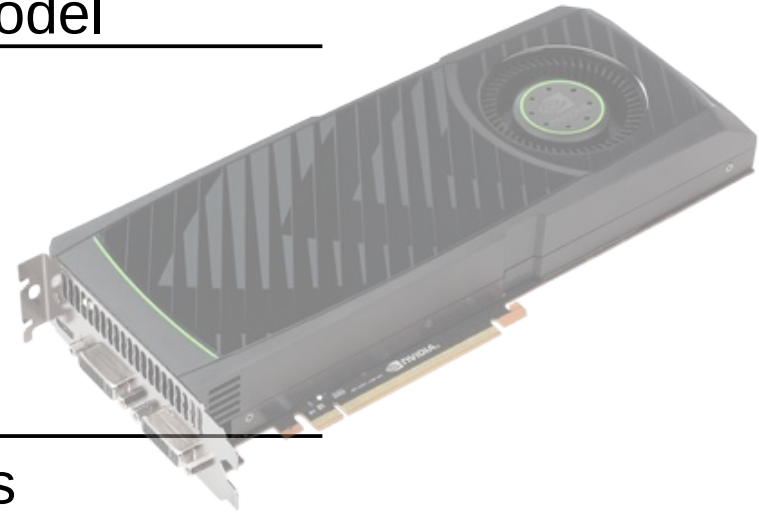
Grid 0



Architecture: **multi-thread** programming model

SIMT microarchitecture

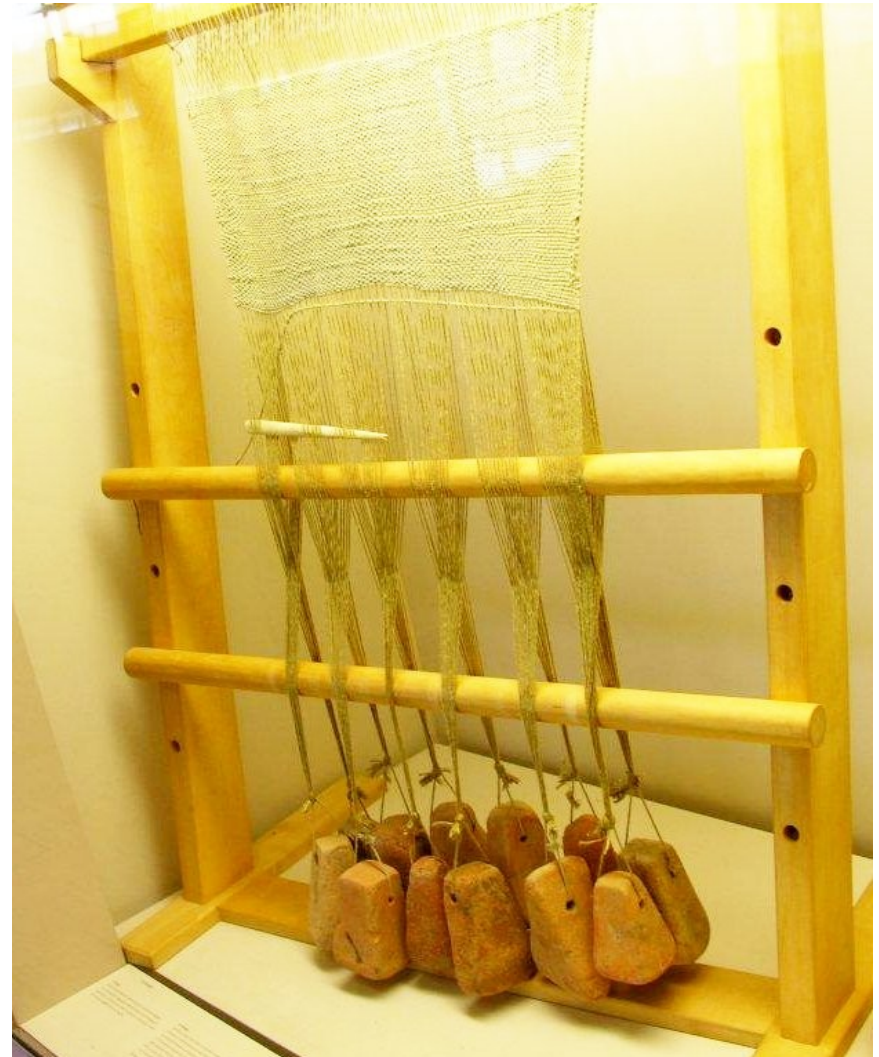
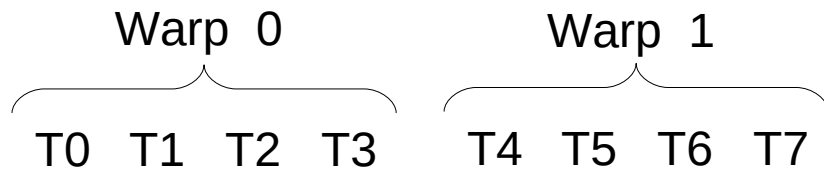
Hardware datapaths: **SIMD** execution units



Hardware

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:

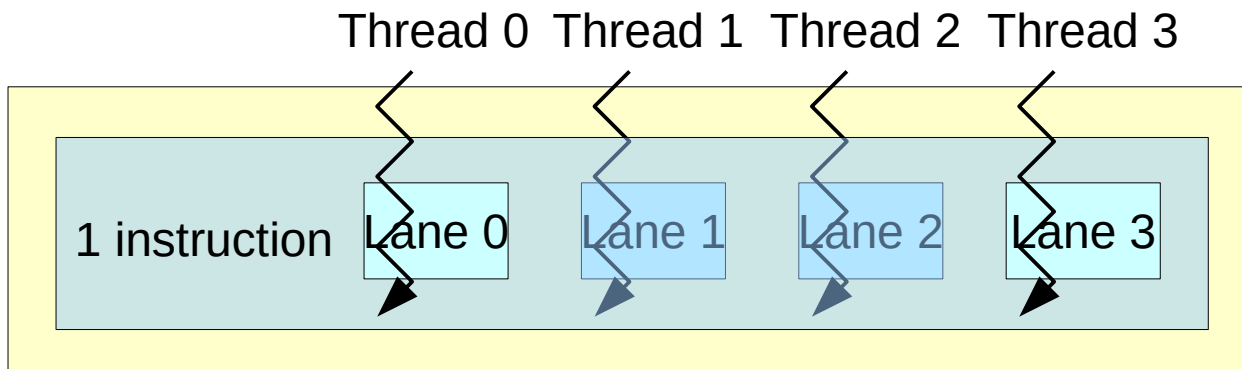
| | Warp | | | |
|-------------------------------------|------|----|----|----|
| | T0 | T1 | T2 | T3 |
| <code>x = 0;</code> | | | | |
| <code>// Uniform condition</code> | | | | |
| <code>if(a[x] > 17) {</code> | 1 | 1 | 1 | 1 |
| <code> x = 1;</code> | | | | |
| <code>}</code> | | | | |
| <code>//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

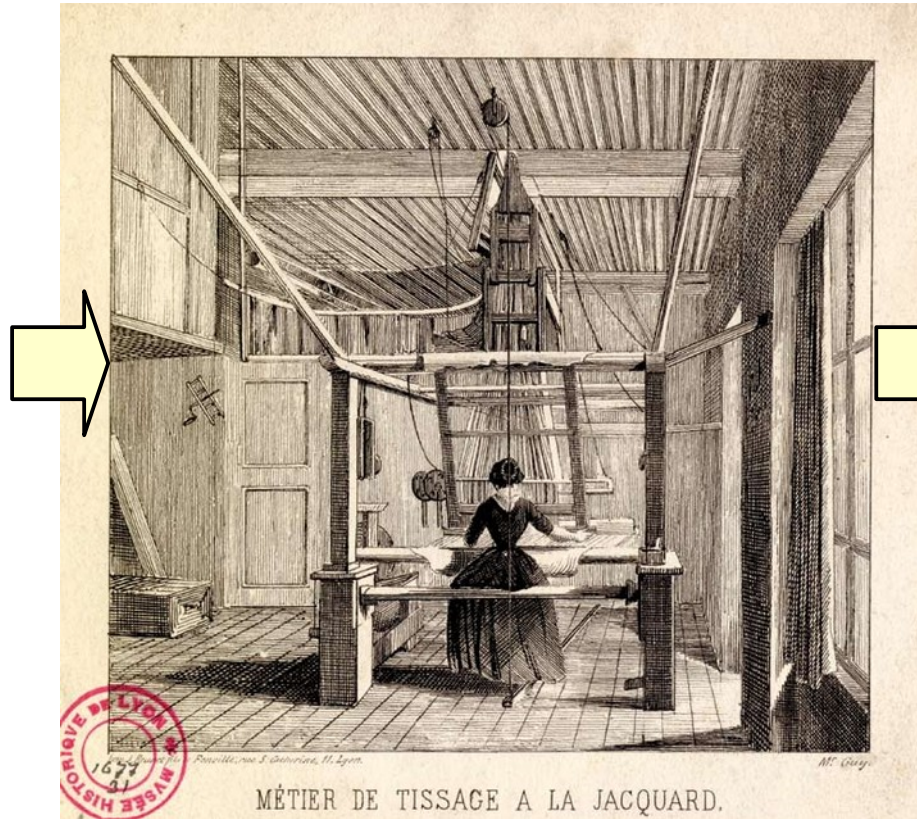
| | | | | | |
|--|-----------------------------|----------|----------|----------|----------|
| <pre>x = 0; // Uniform condition if(a[x] > 17) { x = 1; }</pre> | if condition: a[x] > 17? | Warp | | | |
| | | T0 | T1 | T2 | T3 |
| | | 1 | 1 | 1 | 1 |
| | Execution mask: | 1 | 1 | 1 | 1 |
| | | | | | |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

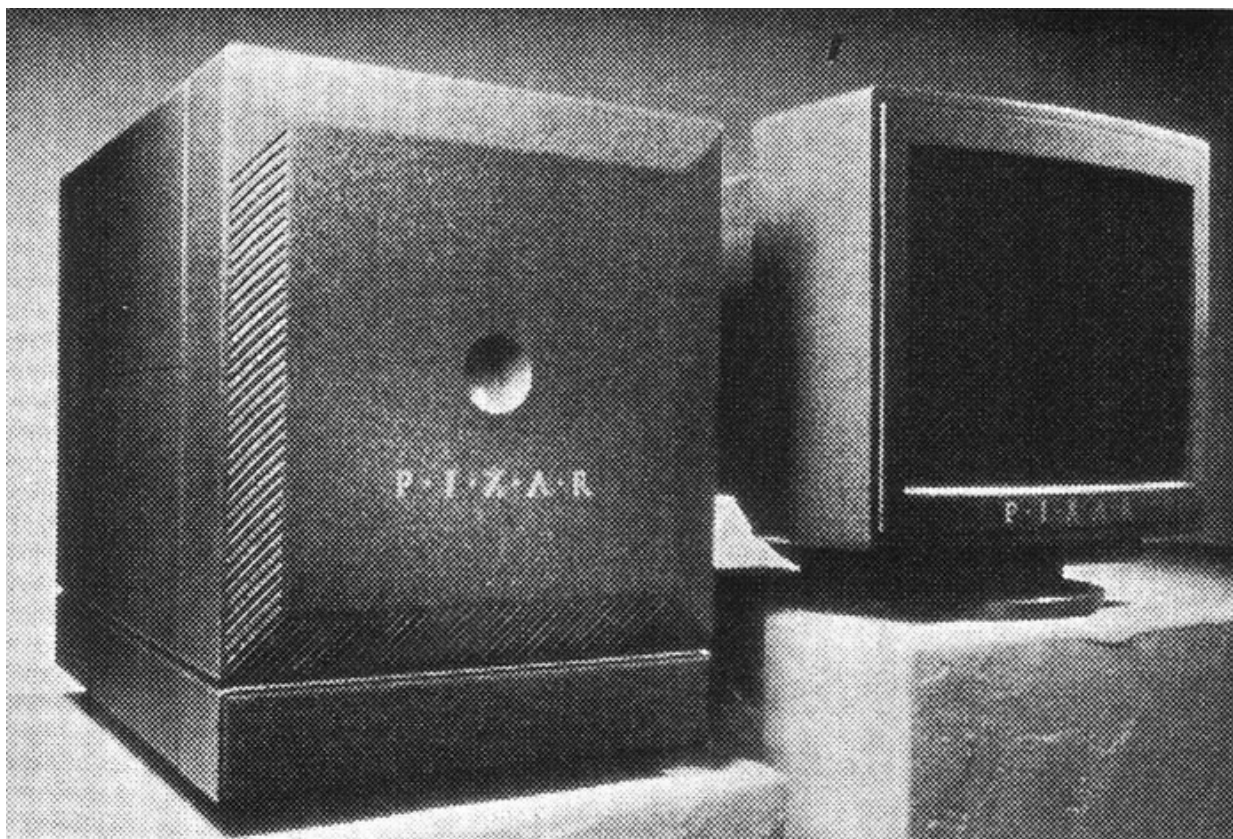
- Running SPMD software on SIMD hardware
 - Context: software and hardware
 - The control flow divergence problem
- Stack-based control flow tracking
 - Stacks
 - Counters
- Path-based control flow tracking
 - The idea: use PCs
 - Implementation: path list
 - Applications

Quizz

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

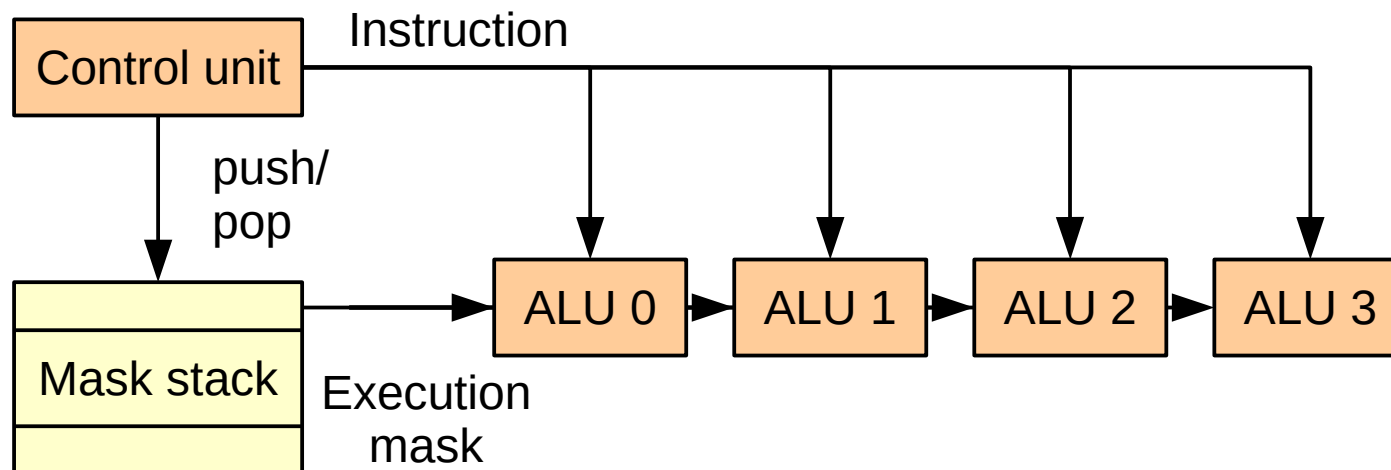
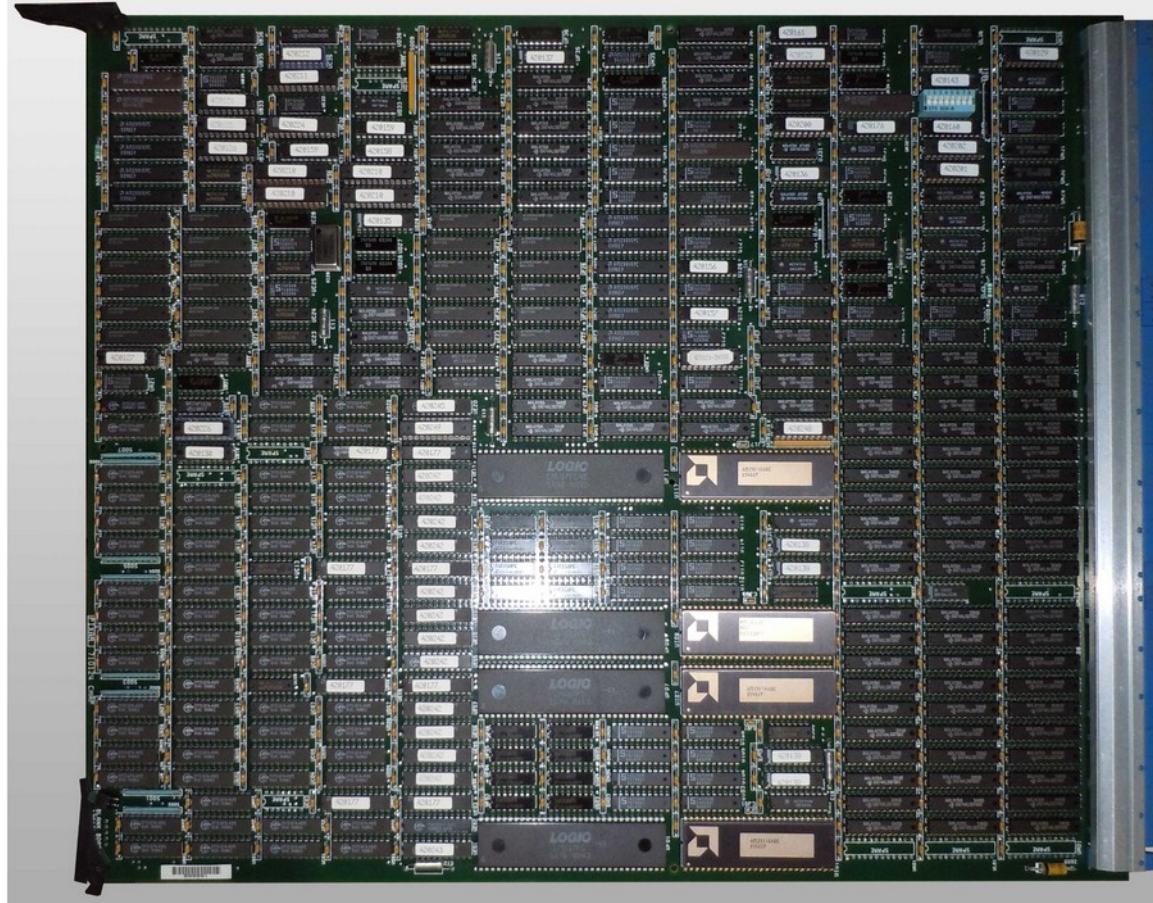
Answer: Pixar!

- 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**

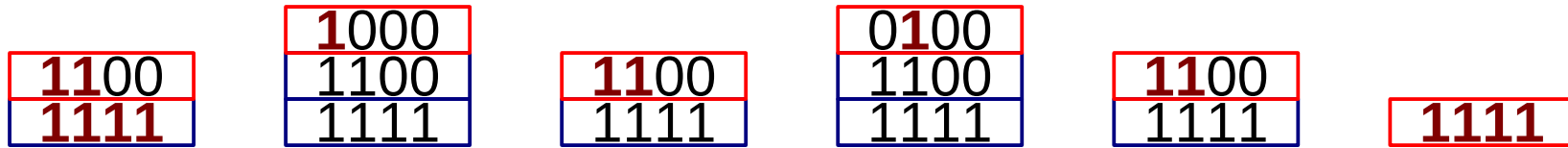
1111

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

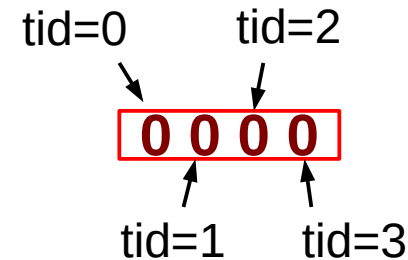
```
    }
```

dec

```
}
```

Counters

1 (in)activity counter / thread



| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 2 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

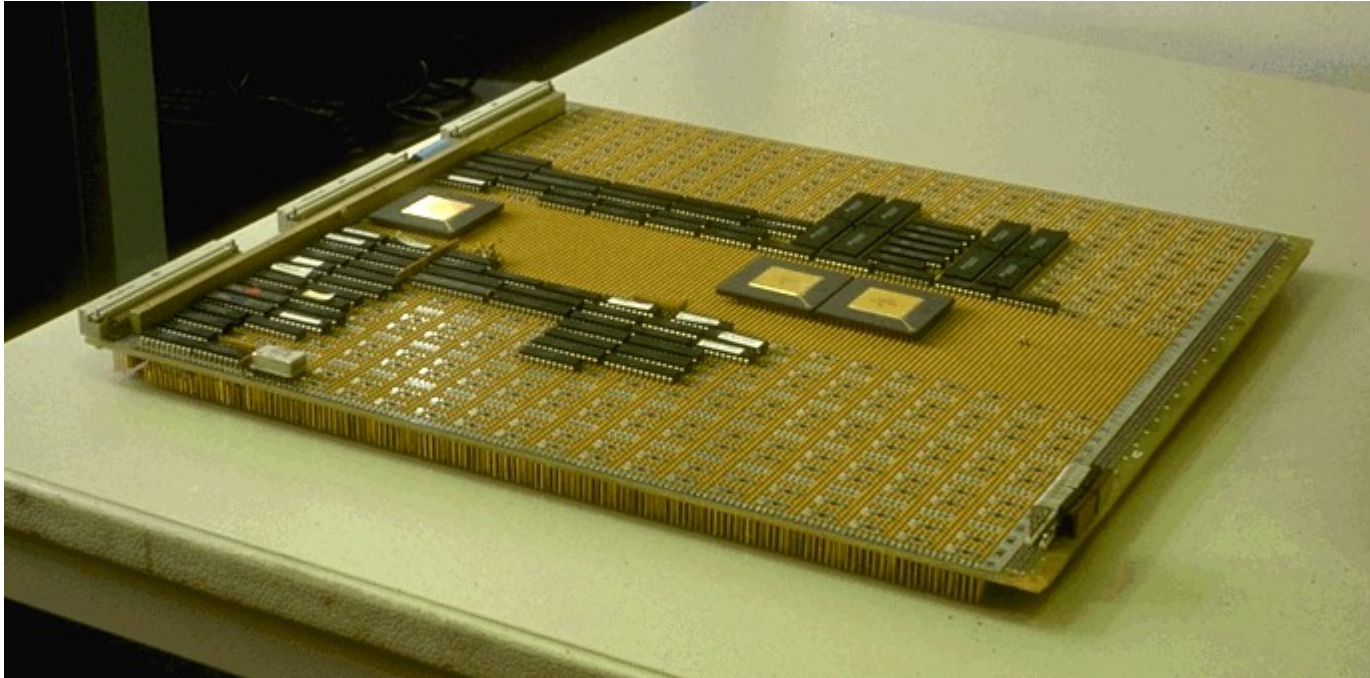
| | | | |
|---|---|---|---|
| 1 | 0 | 2 | 2 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 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

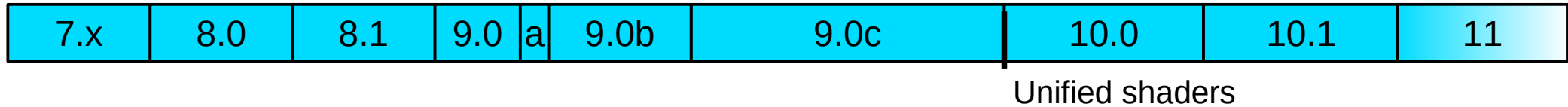


Outline

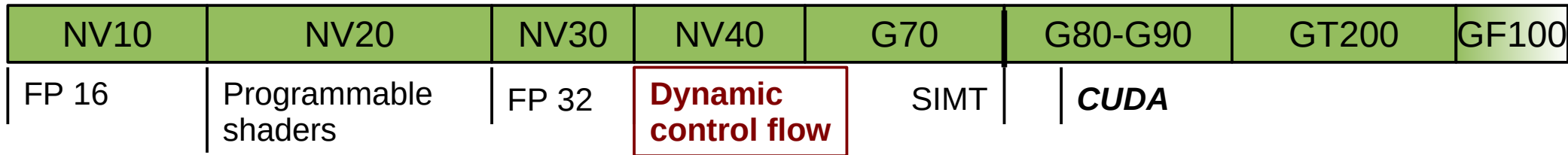
- Running SPMD software on SIMD hardware
 - Context: software and hardware
 - The control flow divergence problem
- Stack-based control flow tracking
 - Stacks, counters
 - A compiler perspective
- Path-based control flow tracking
 - The idea: use PCs
 - Implementation: path list
 - Applications
- Software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

Back to ancient history

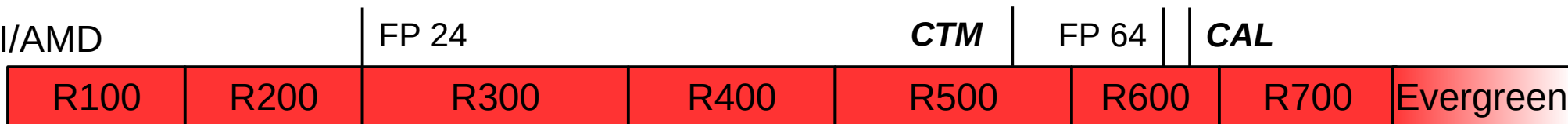
Microsoft DirectX



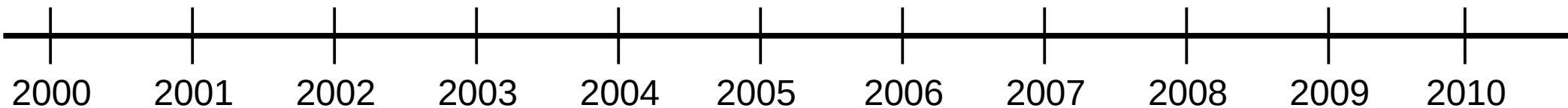
NVIDIA



ATI/AMD



GPGPU traction



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 jal jr syscall | bar bra brk brkpt cal cont kil pbk pret ret ssy trap .s | bar bpt bra brk brx cal cont exit jcal jmx kil pbk pret ret ssy .s | jmp if iff else endif do while break cont halt msave mrest push pop | jmp if else endif case while break cont halt call return fork | jump loop endloop rep endrep breakloop breakrep continue | 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 | 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 |

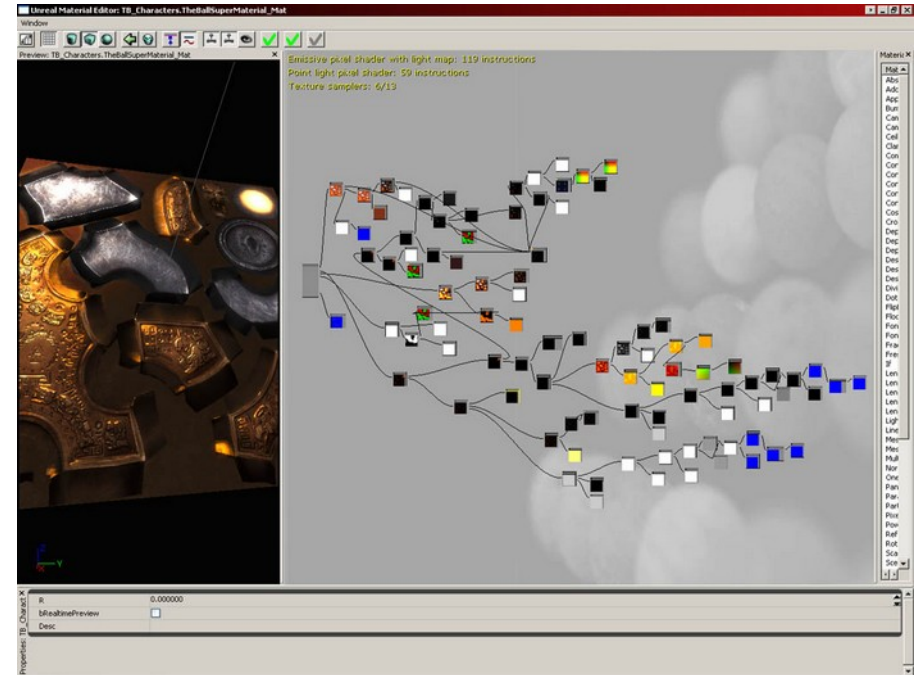
Control instructions in some CPU
and GPU instruction sets

- 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.

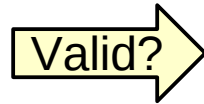
Infinite loop, lock never released.

- Stack-based SIMT divergence control can cause starvation!

Issues of stack-based implementations

- Are all control flow optimizations valid in SIMT?

`f();`



```
if(c)
    f();
else
    f();
```

- 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

Break!

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

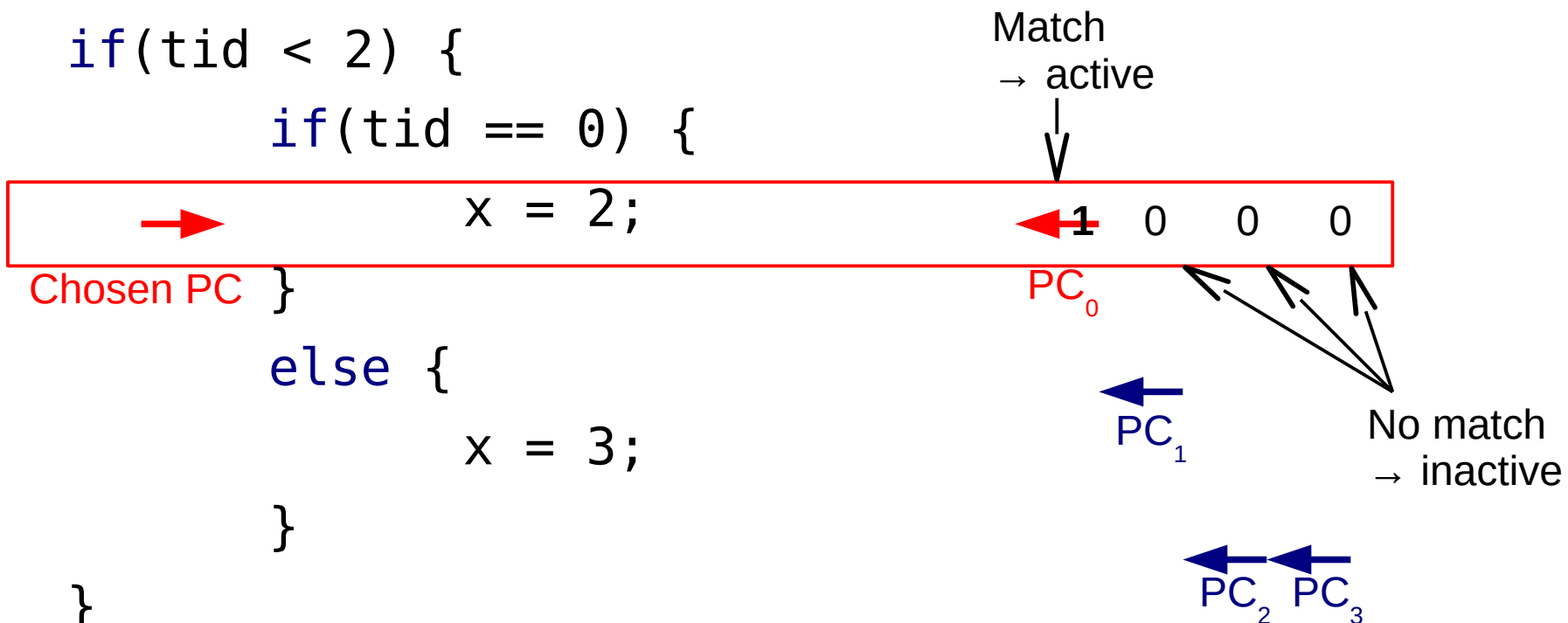
With 1 PC / thread

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

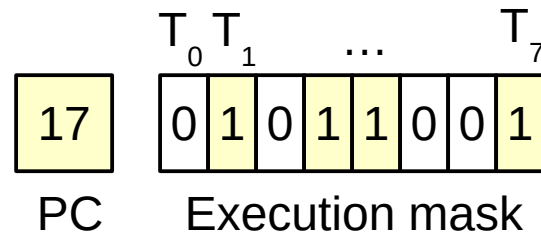


Mask stacks vs. per-thread PCs

- 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



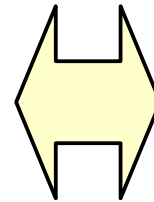
- The mask encodes the **set of threads** that have this PC

$\{ T_1, T_3, T_4, T_7 \}$ 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 |

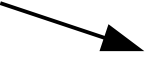
List of paths

- 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

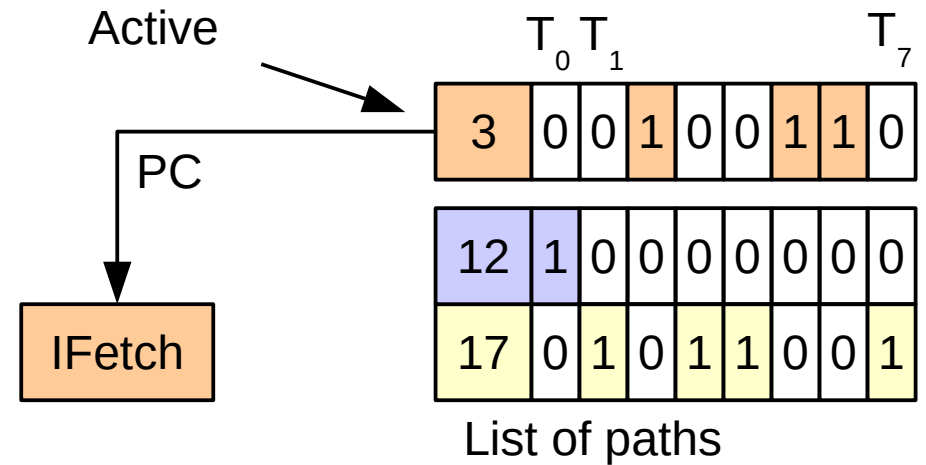


| | T_0 | T_1 | | | | | | T_7 |
|----|-------|-------|---|---|---|---|---|-------|
| 3 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 12 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

List of paths

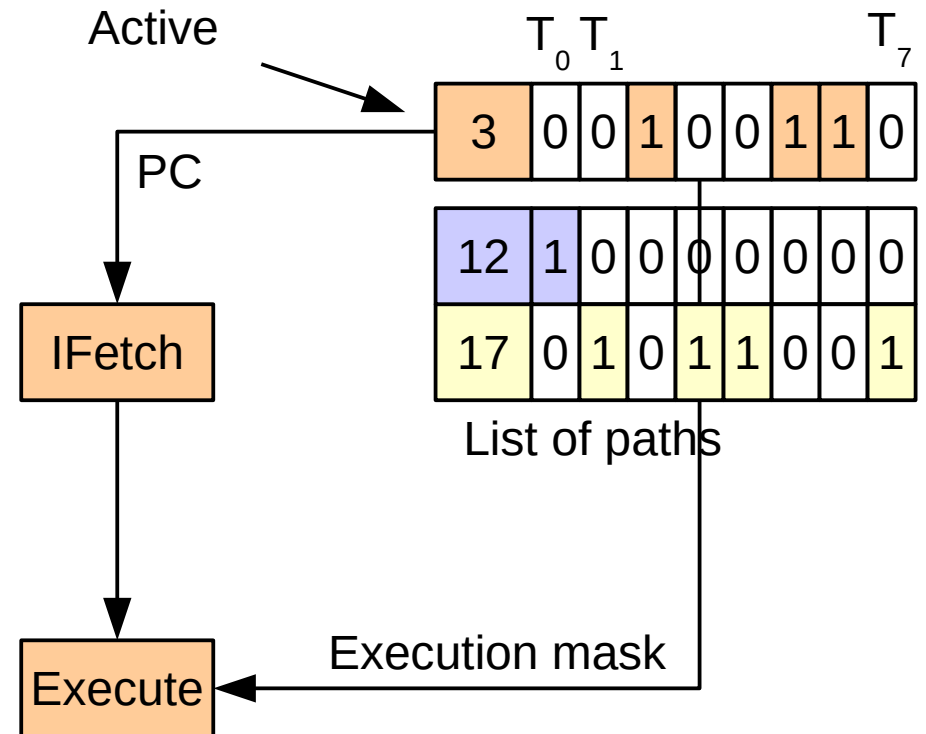
Pipeline overview

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



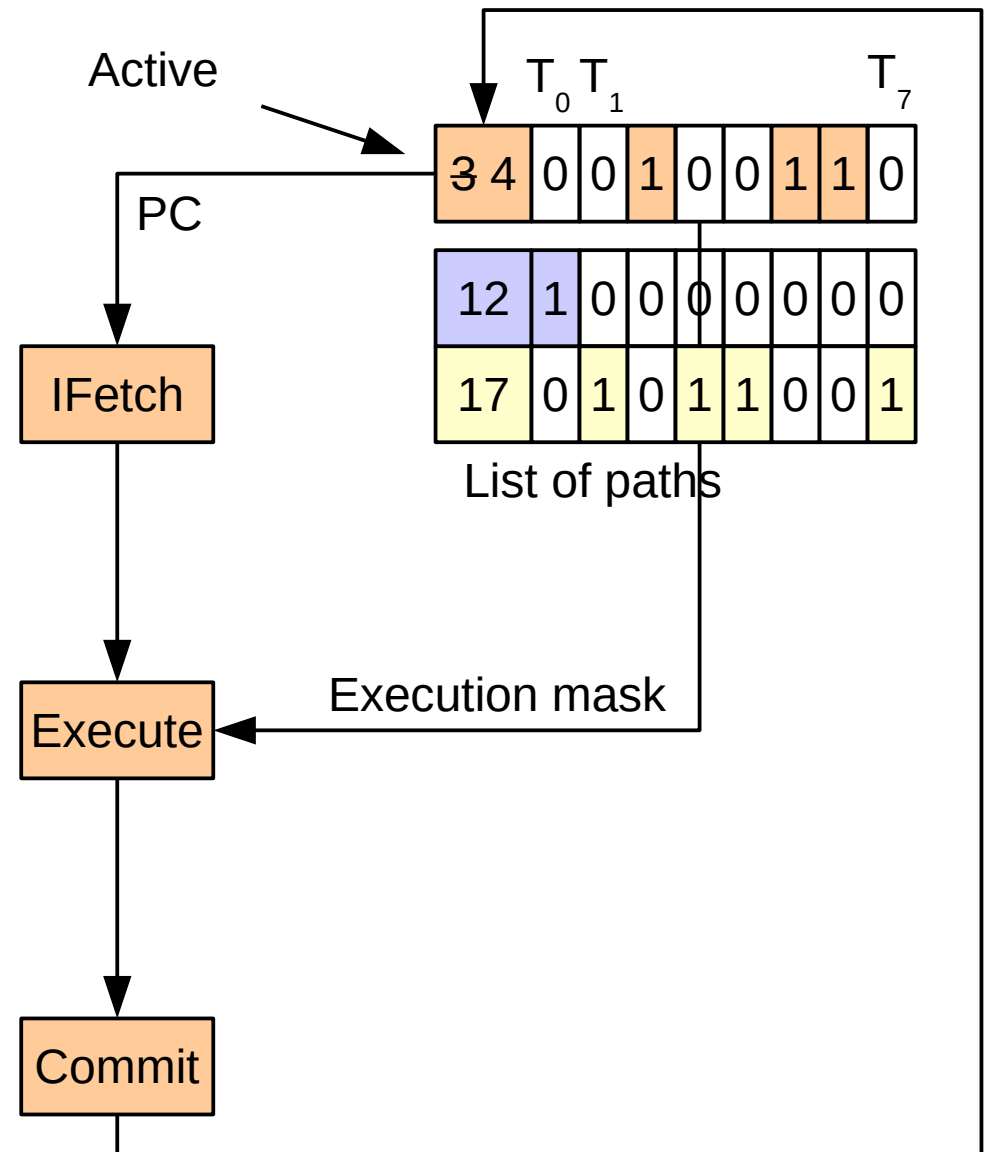
Pipeline overview

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



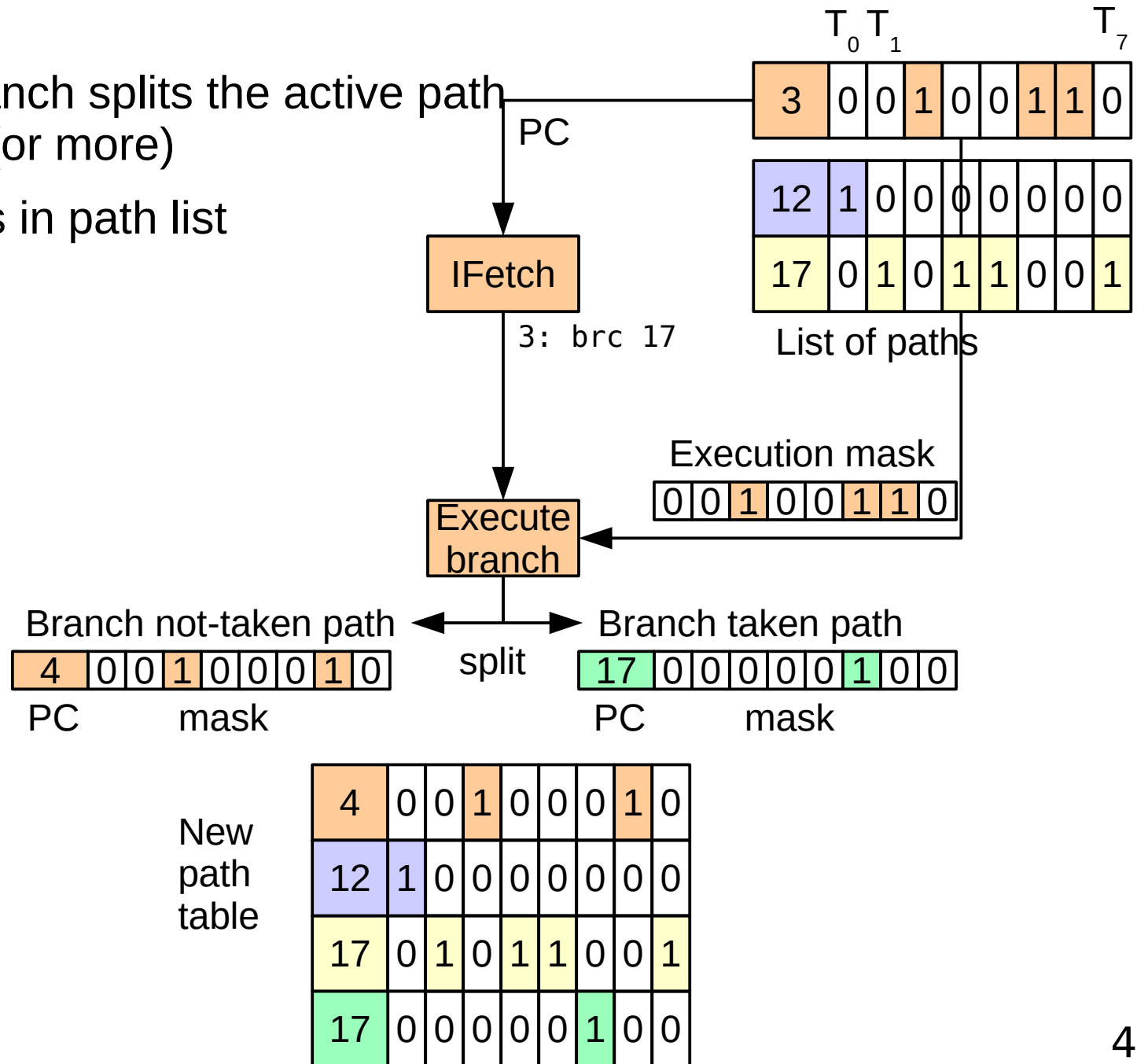
Pipeline overview

- 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 path



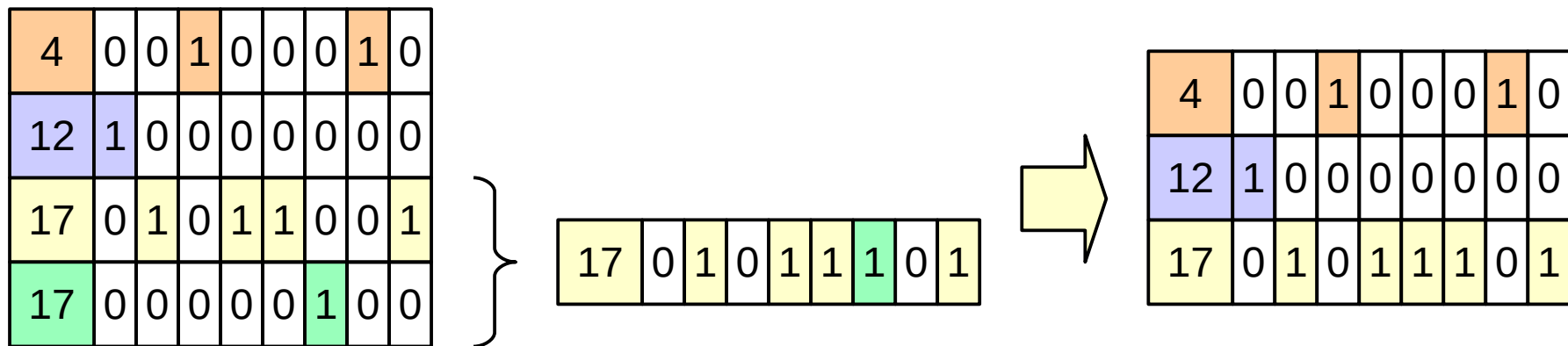
Divergent branch is path insertion

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



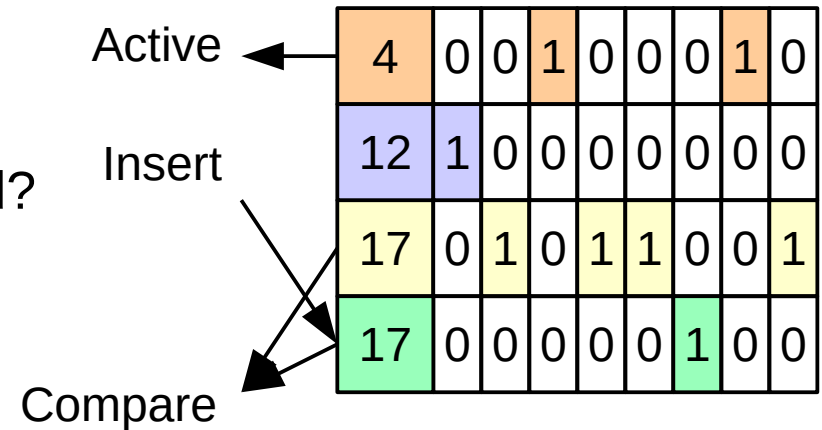
Convergence is path fusion

- 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

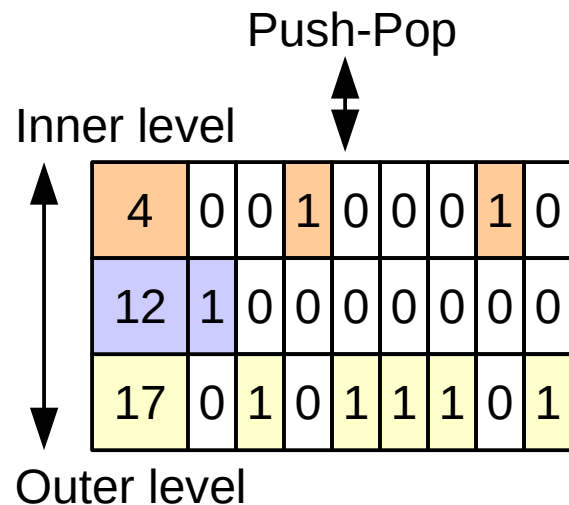
- Degrees of freedom
 - Which path is the active path?
 - At which place are new paths inserted?
 - When and where do we check for convergence?



- 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



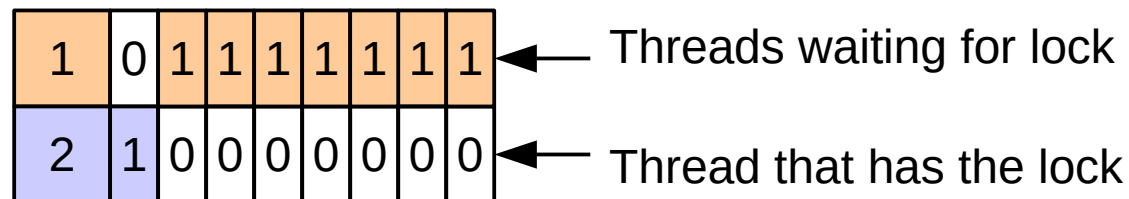
```
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}
// Divergent conditions
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    }
    else {
        x = 3;
    }
}
```

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

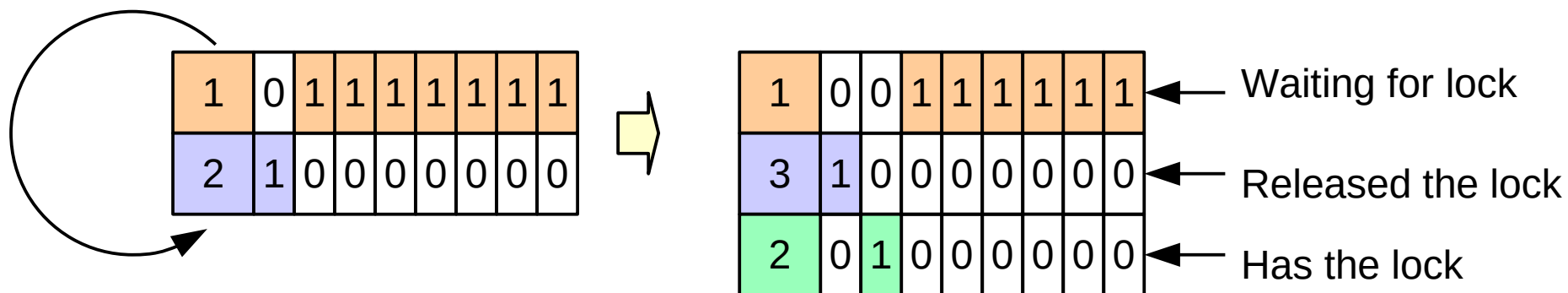
Breadth-first graph traversal

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

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



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



- Drawback: may delay convergence

Example: Nvidia Volta (2017)

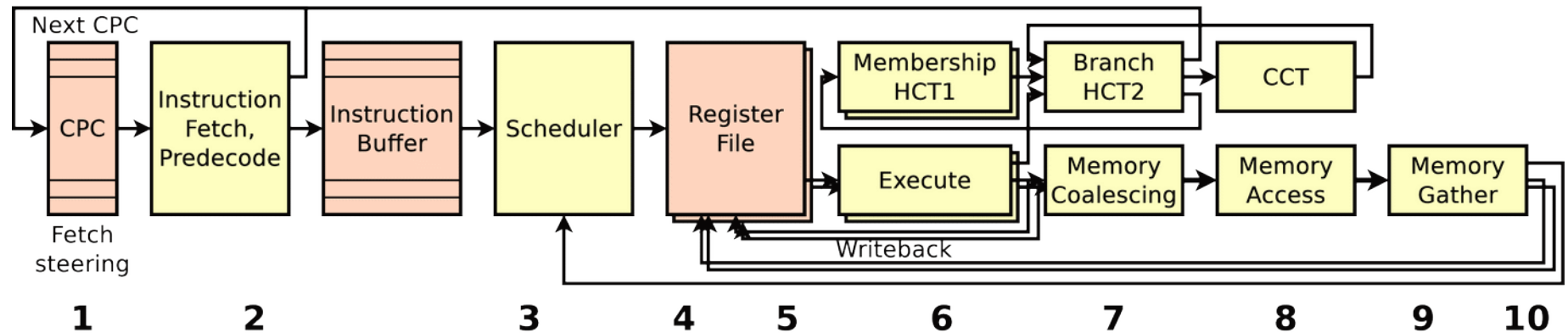
Supports independent thread scheduling inside a warp

- Threads can synchronize with each other inside a warp
- Diverged threads can run barriers (as long as all threads eventually reach a barrier)



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



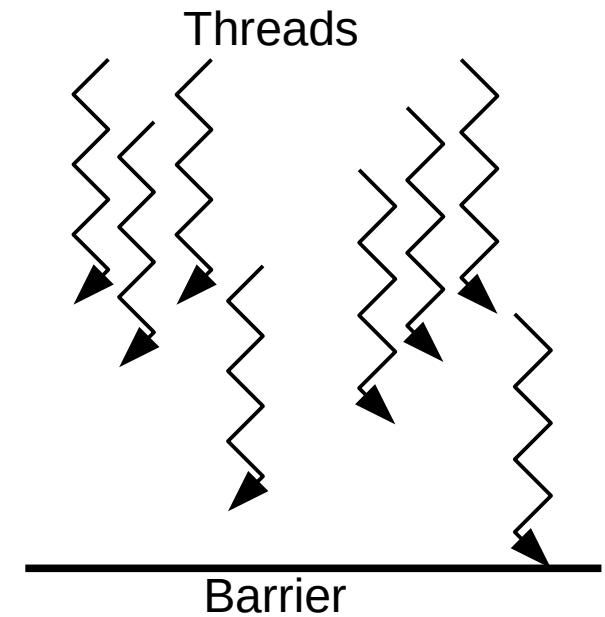
Programmer's view

- 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`
 - Code that is hard to indent!