Exceptional Control Flow

CSCI 237: Computer Organization
35th Lecture, Wednesday, December 2, 2025

Kelly Shaw

Administrative Details

- Read CSAPP Ch. 8.1-8.2
- Lab #6 due Friday at 5pm
- TA Feedback form
  - https://forms.gle/mwaWEUy46iHT4MT37
- Review session poll
  - Either Thursday (12/11) or Friday (12/12)
- Colloquium Friday at 2:35pm
  - Xiwei Xuan, UC Davis
  - Toward Responsible AI through Efficient Data Science and Learning
- Talk about final exam on Friday

Last Time

- Processes
- Threads

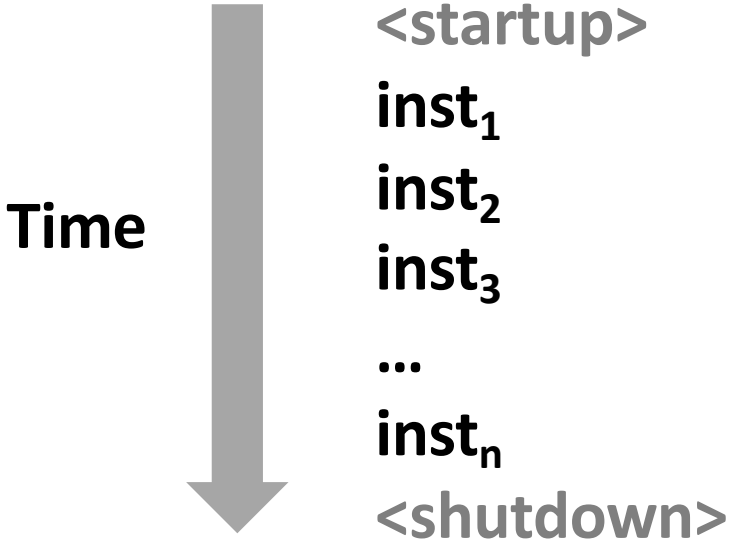
Today

- Exceptional Control Flow
- Exceptions
- Common computer organizations

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow



Altering the Control Flow

- Two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

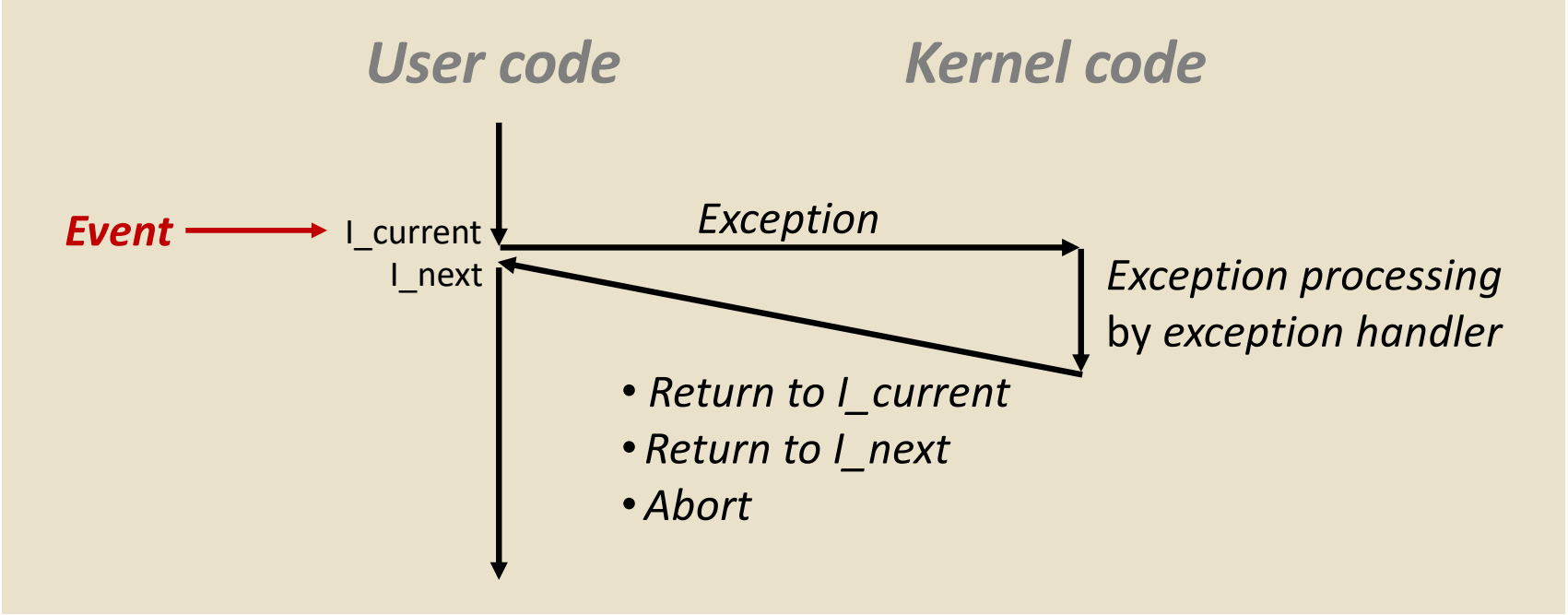
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software, communicate between processes
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library, allows jumps to arbitrary locations
    - Used for exception handling in C (setjmp() for try, longjmp() for throw)

Today

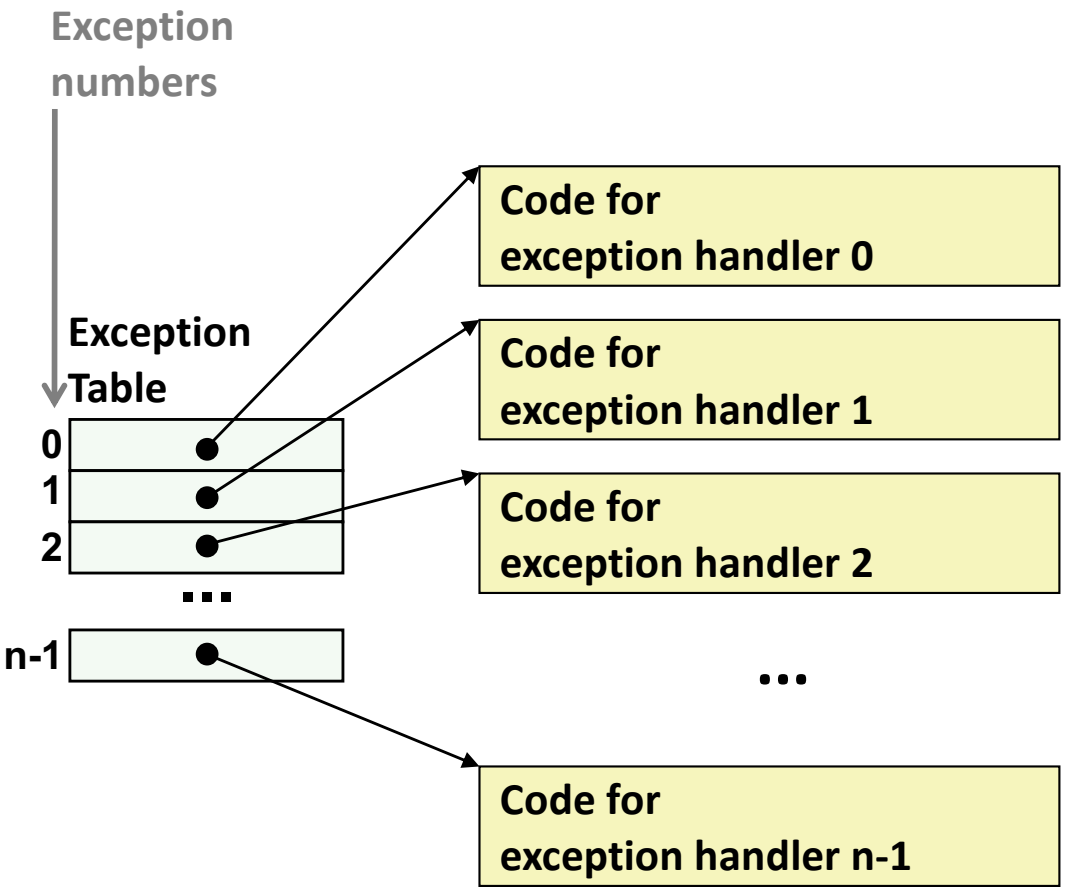
- Exceptional Control Flow
- Exceptions
- Common computer organizations

Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

System Calls

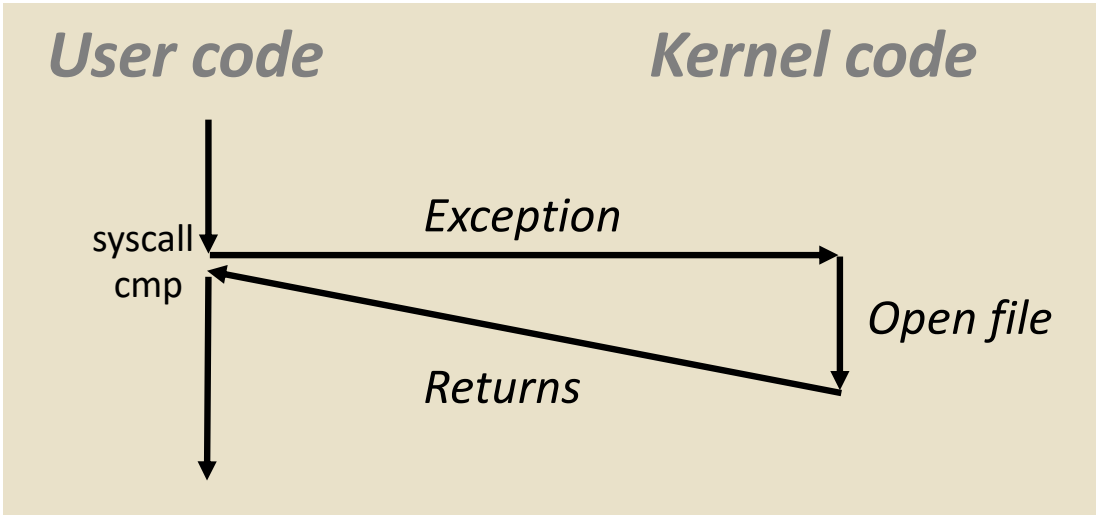
- How user code asks OS to do something on its behalf
- Each x86-64 system call has a unique ID number
- Examples:

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



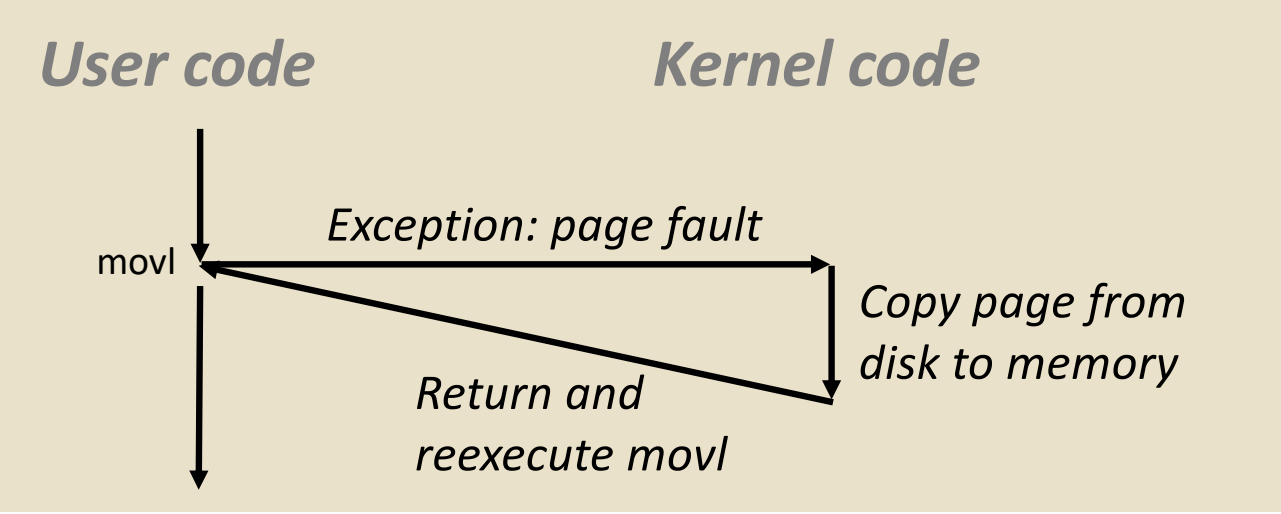
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

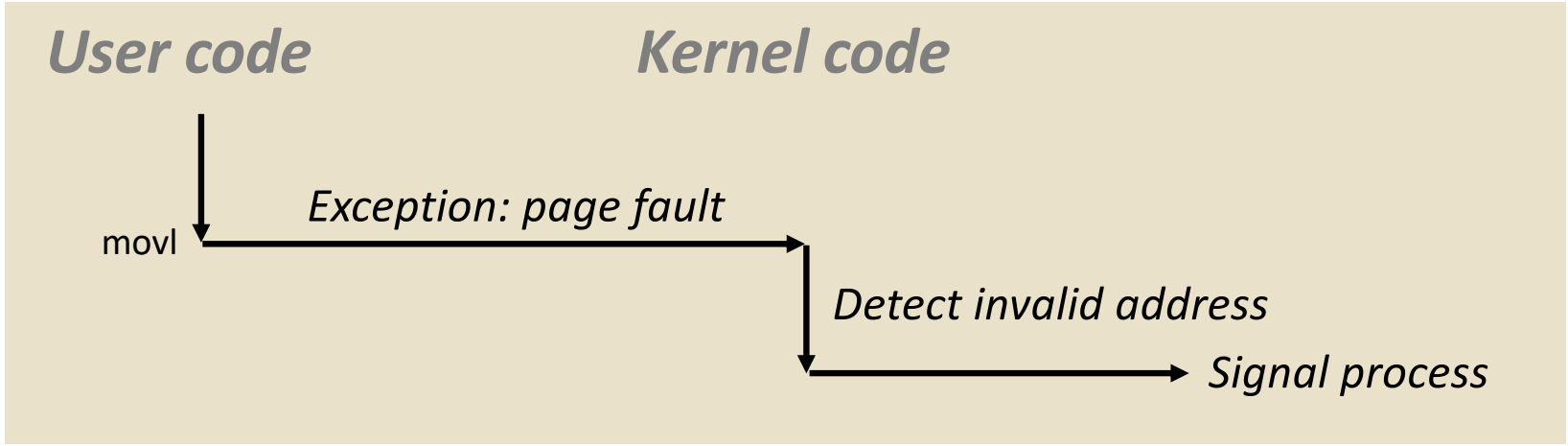
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



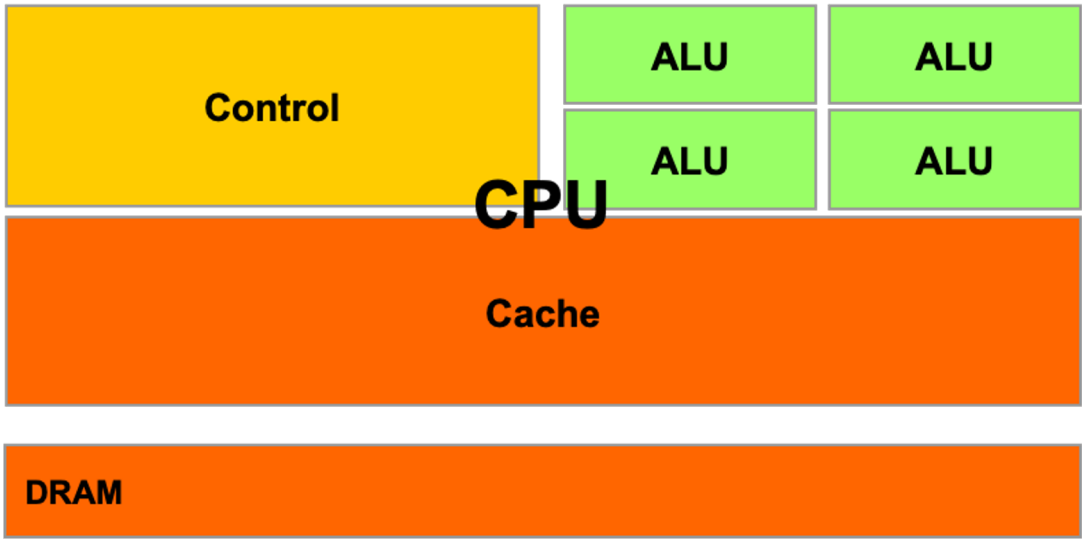
- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

Today

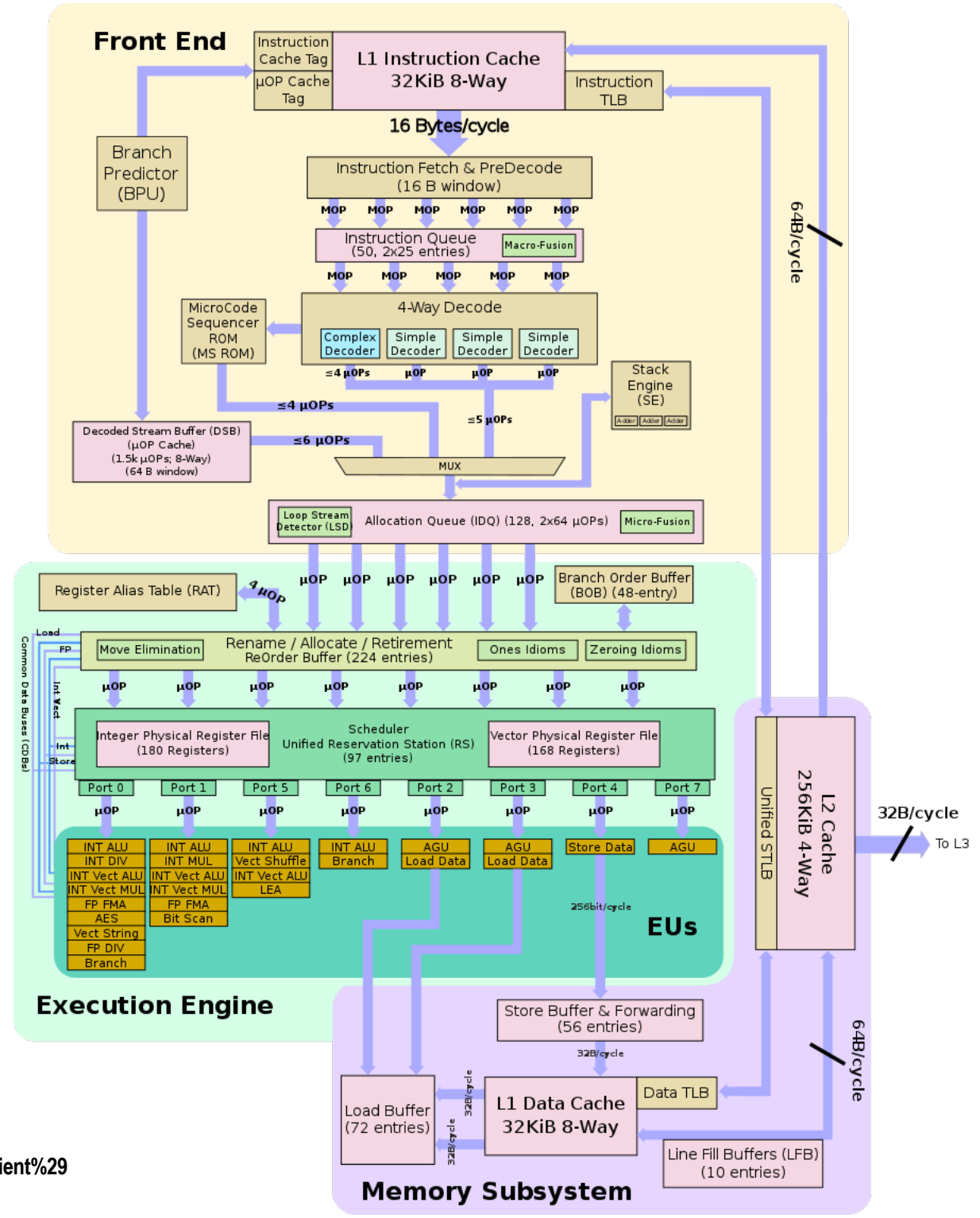
- Exceptional Control Flow
- Exceptions
- Common computer organizations

CPUs: Latency Oriented Design

- Optimize for single thread performance
- High clock frequency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency

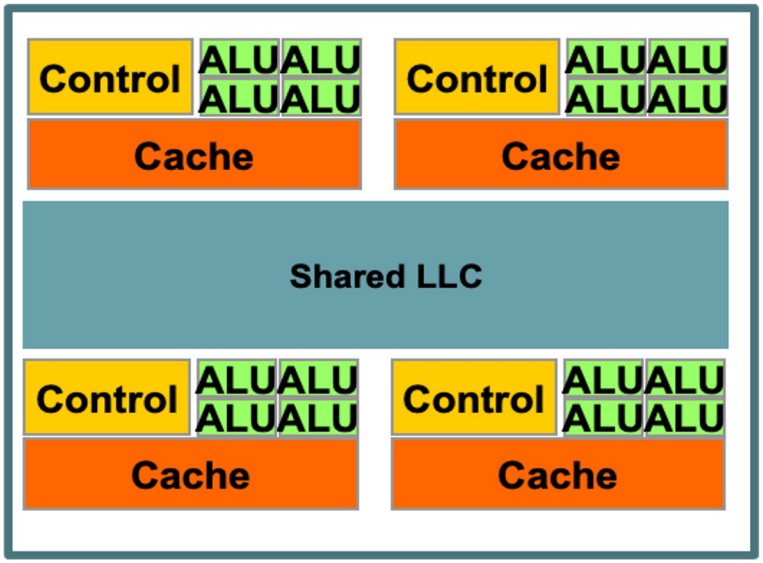


Intel Skylake



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_%28client%29

Move to Multi-Core and Many Core Systems

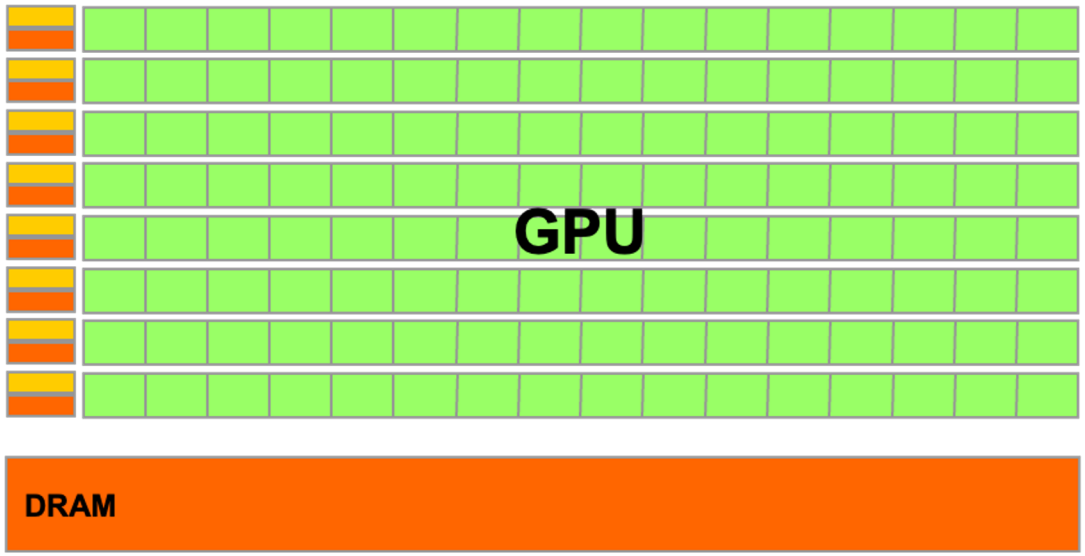


Multi-core

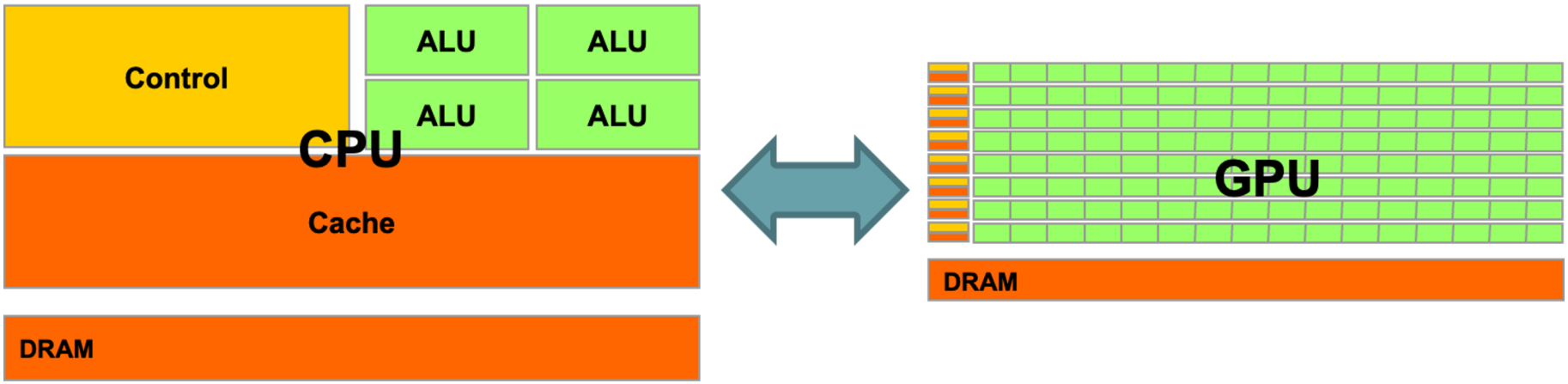
Many-Core

GPUs: Throughput Oriented Design

- Moderate clock frequency
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



Applications Benefit from Both CPU and GPU



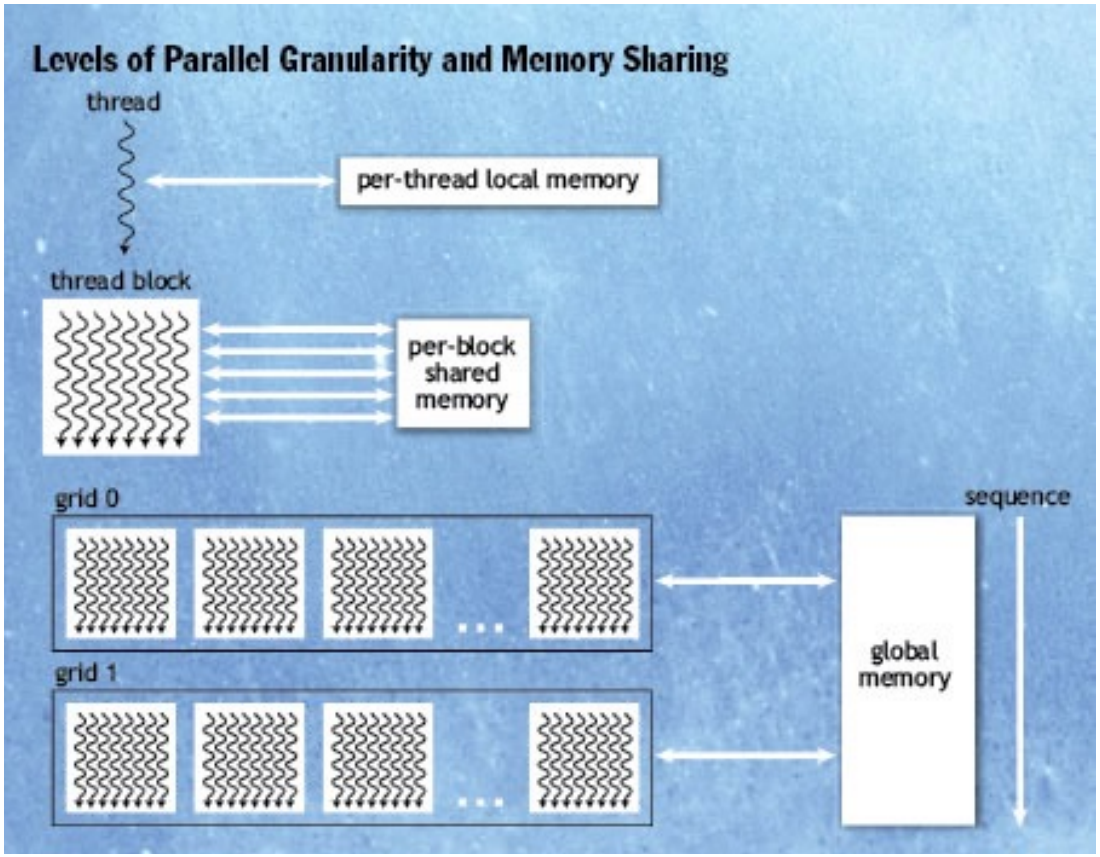
CPUs for sequential parts where latency matters

- CPUs can be 10+X faster than GPUs for sequential code

GPUs for parallel parts where throughput wins

- GPUs can be 10+X faster than CPUs for parallel code

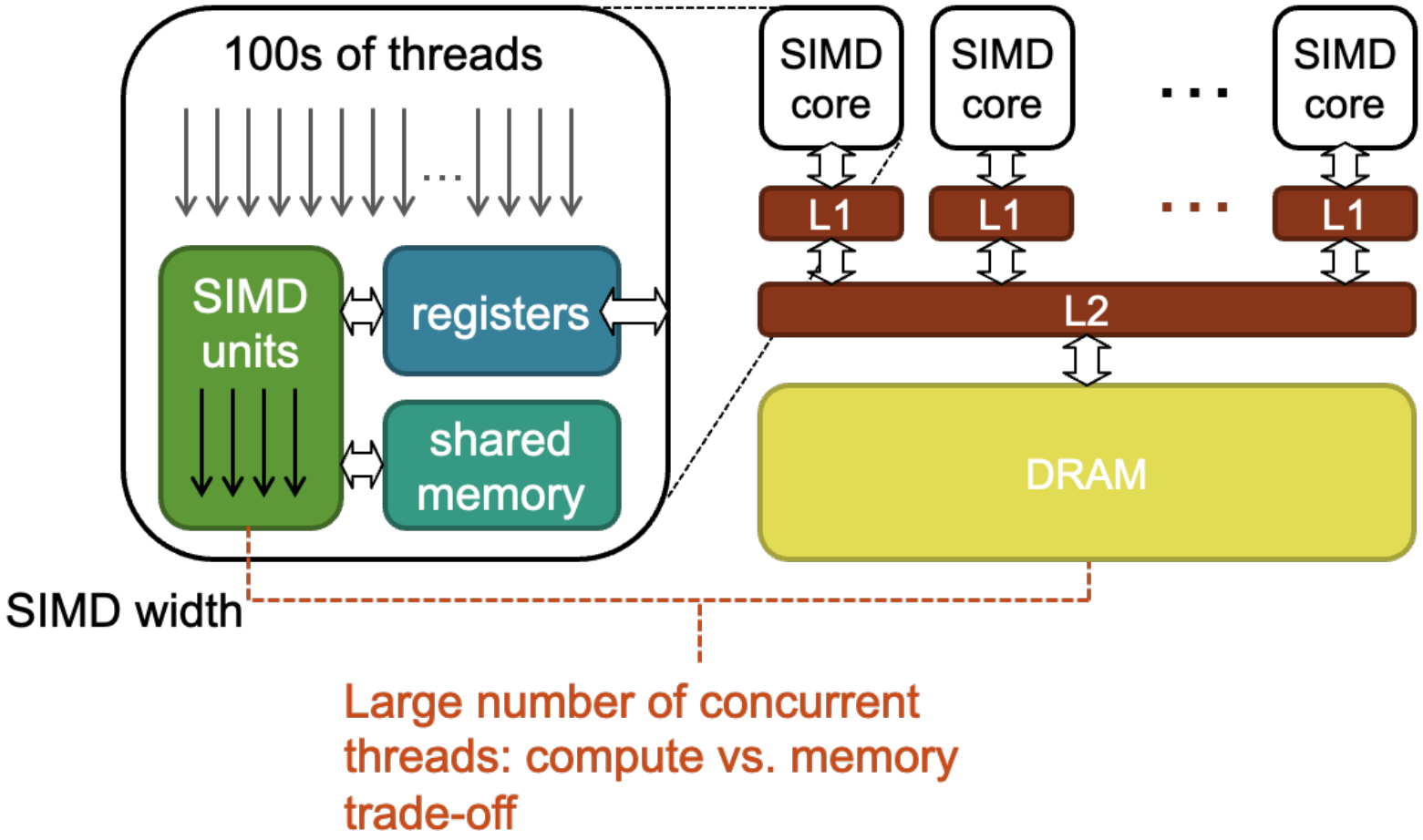
What's So Special About GPUs? Programming



- Thousands of parallel threads executing same code
- Hierarchical thread organization impacts memory sharing
- Data managed by software

J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA". *Queue* (March 2008).

GPUs: A Closer Look



Writing Applications to use GPUs

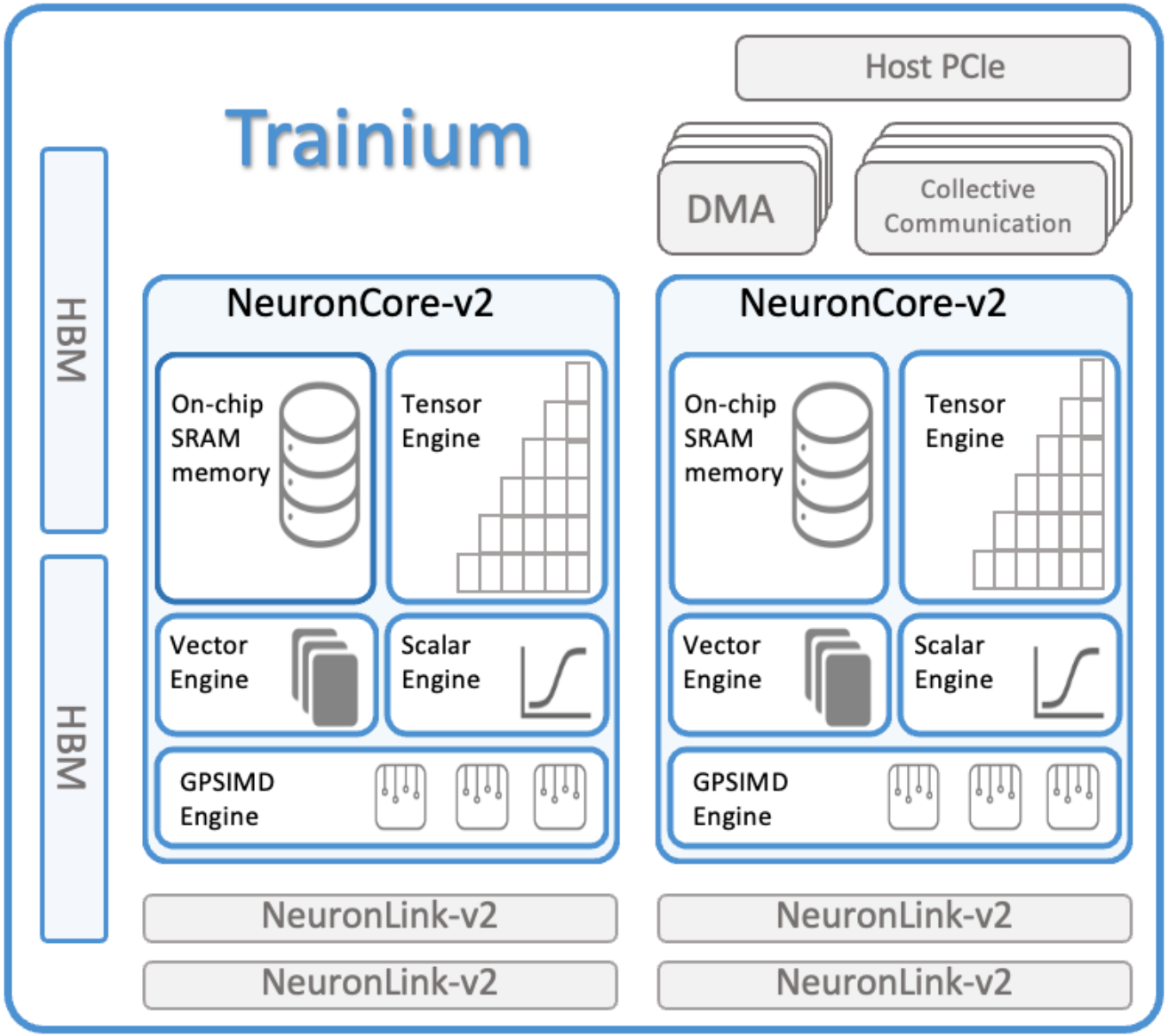
Write code for CPU and for GPU

- CPU is in control
- CPU must transfer data to GPU device memory
- CPU invokes GPU kernel and typically waits for completion
- CPU transfers data from GPU device memory back to CPU memory

Developers job

- Must orchestrate movement of data between CPU and GPU
- Must orchestrate movement of data into special, fast, software managed memory
- Must orchestrate the collaboration of threads and the sharing of fast memory among threads

Accelerators



- Specialized processors geared to speed up specific algorithms
- AWS general purpose machine learning accelerator

https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/arch/neuron-hardware/trainium.html