CMPE110 Lecture 24

Multicore

Heiner Litz

https://canvas.ucsc.edu/courses/12652

Announcements

- Next week
  - Monday: Pat will give exam review session 1
  - Wednesday: Heiner will give Zoom lecture on Multicore/GPU
  - Friday: Heiner will give exam review session 2

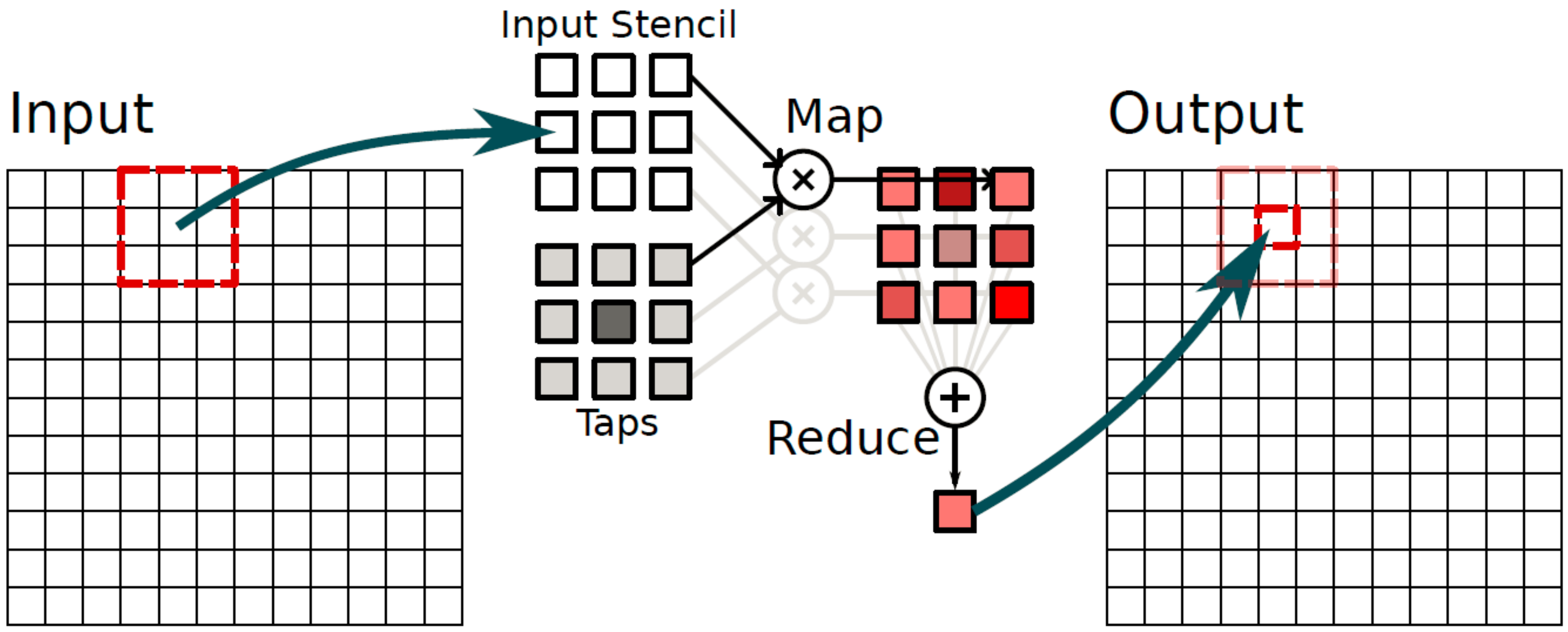
- Office Hours
  - No office hours on Mon 4th and Wed 6th
  - Office hours today 2:30-4pm
  - Office hours Friday 8th, 2:30-4pm

Additional Reading Material

- I/O: Computer Architecture, A Quantitative Approach, 4th Edition, Chapter 6, Storage Systems
- Multicore: Computer Organization and Design, Chapter 6, Parallel Processors, 6.1 – 6.5

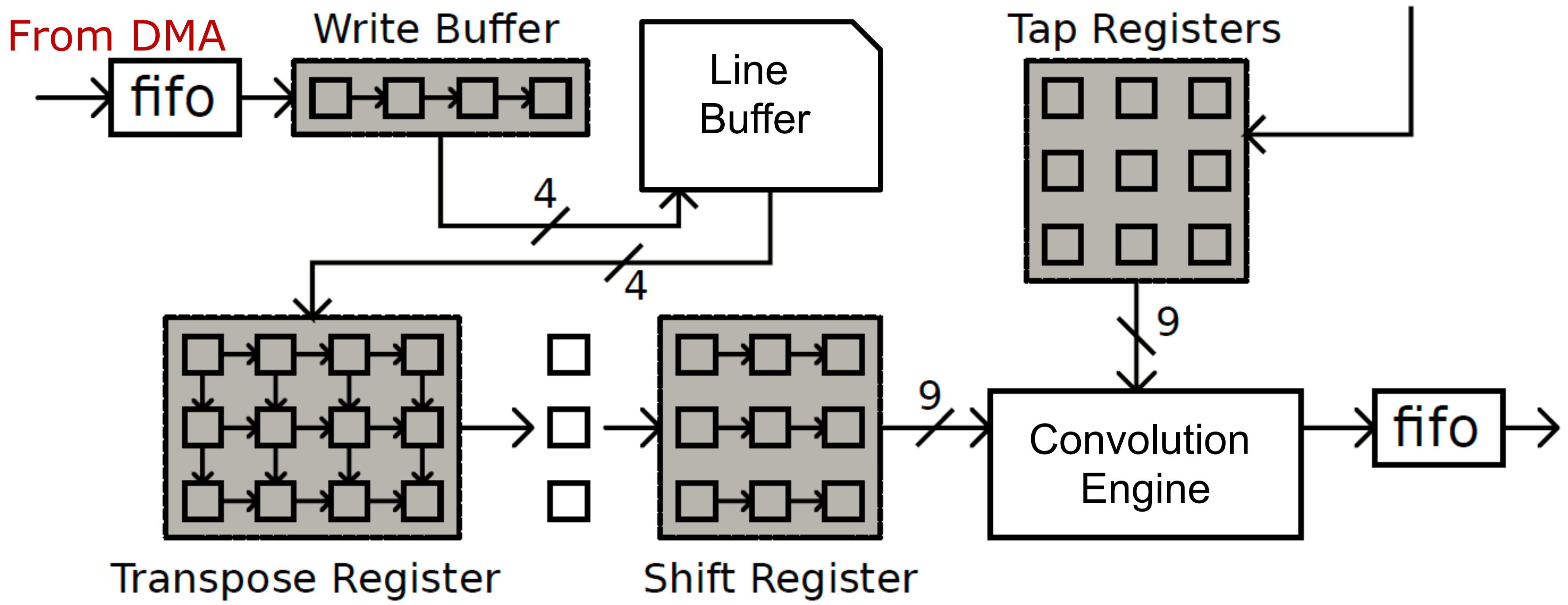
Review

Convolution Engine



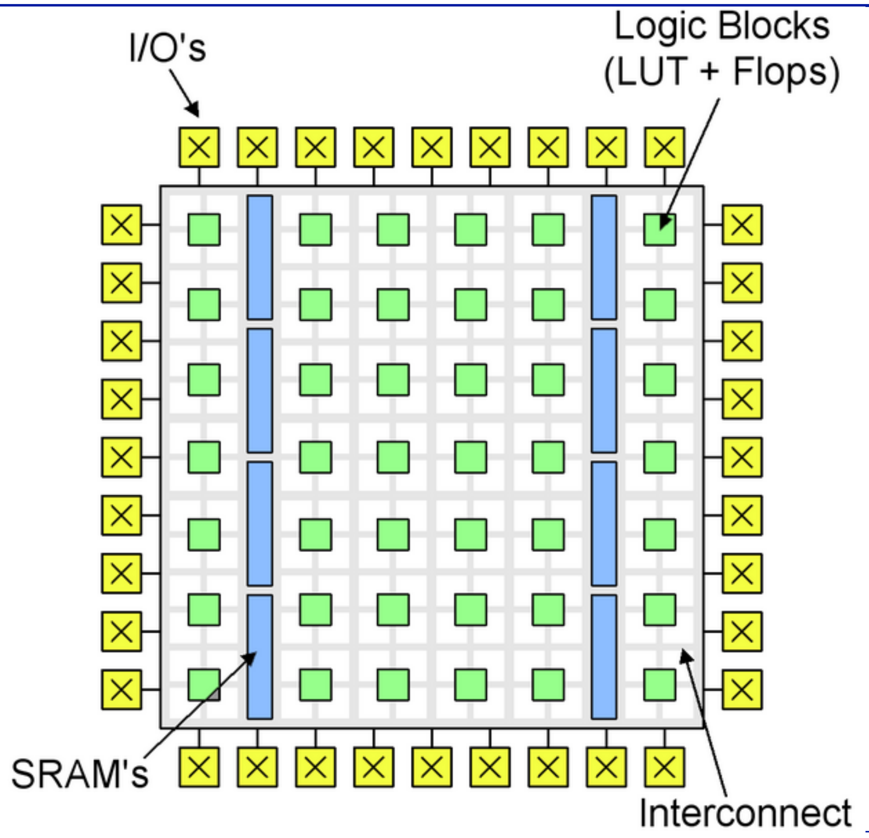
- How many rows of data do we need to buffer?
- How much new data do we need per pixel?

Convolution Datapath

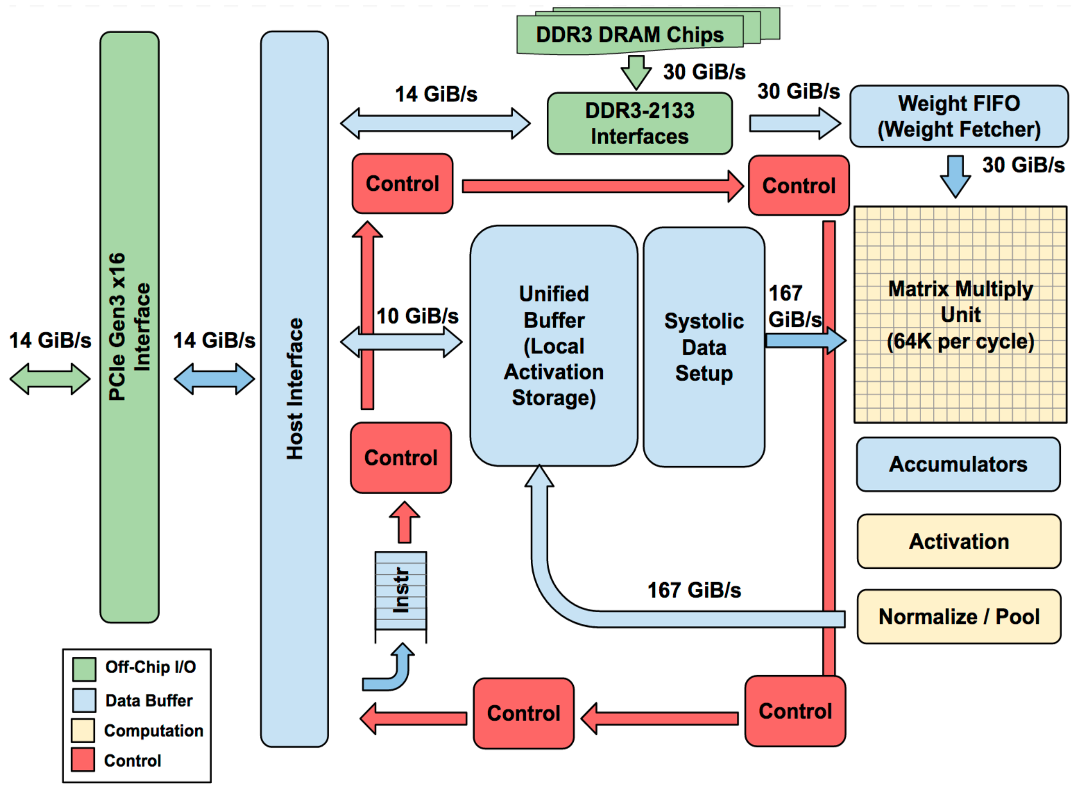


- How many pixels per cycle?

Example - FPGAs

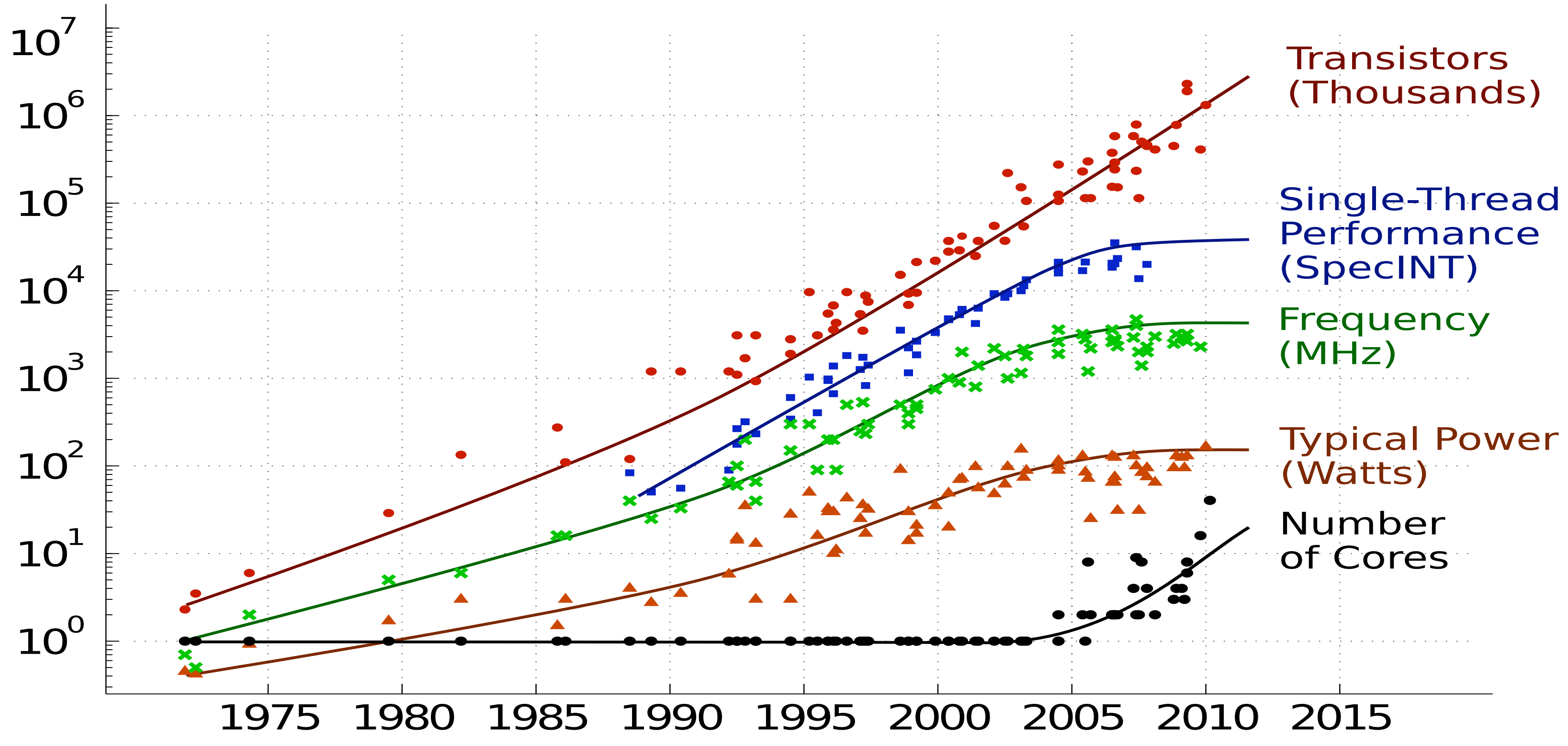


Google TPU


DDR3 DRAM Chips
30 GiB/s
14 GiB/s
DDR3-2133 Interfaces
30 GiB/s
Weight FIFO (Weight Fetcher)
30 GiB/s
Control
Control
14 GiB/s
PCIe Gen3 x16 Interface
14 GiB/s
Host Interface
10 GiB/s
Unified Buffer (Local Activation Storage)
Systolic Data Setup
167 GiB/s
Matrix Multiply Unit (64K per cycle)
Control
Accumulators
Activation
Instr
167 GiB/s
Normalize / Pool
Control
Control
Off-Chip I/O
Data Buffer
Computation
Control

Multicore

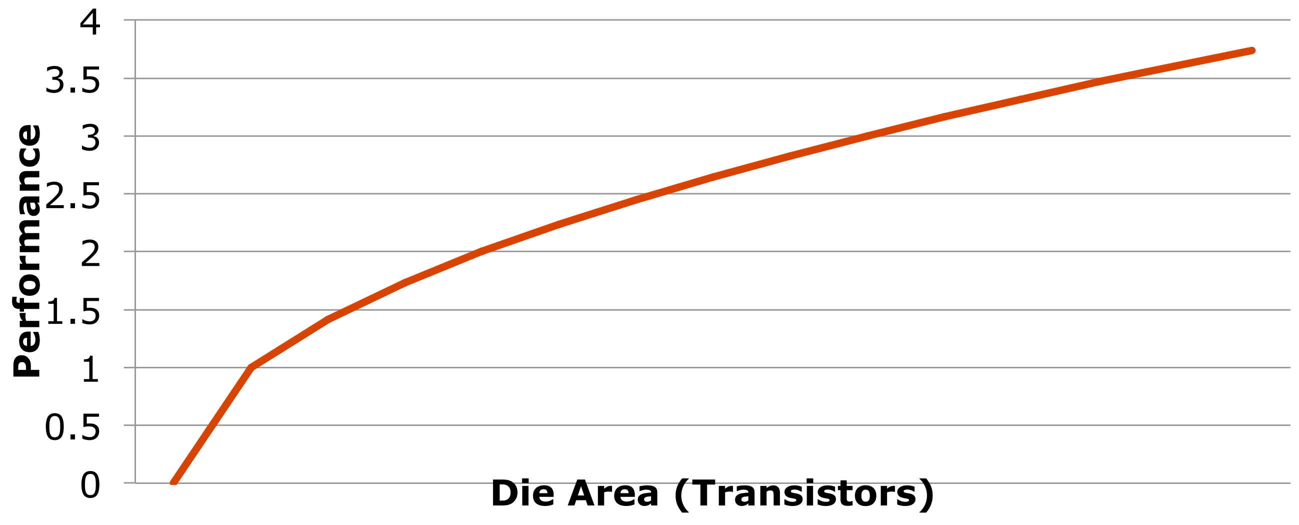
Sequential Processor Performance has Plateaued



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

Why does Multicore help?

- Increasing core count increases power, right?
- Pollack's rule: Performance increase due to microarchitecture improvements equals sqrt() of complexity



Parallel Computing

- Goal: higher performance from multiple processors
  - Multiprocessors ➔ multi-core
  - Scalability, availability, power efficiency

- Challenge
  - Making effective use of parallel hardware
- Process-level parallelism (Multi-process)
  - High throughput for independent jobs
- Parallel program (Multi-threading)
  - Single program run on multiple processors

What We've Already Covered

Parallelism and ISA

- Instruction Level Parallelism (ILP)
- Vector parallelism (SIMD or data-level)

Parallelism and memory hierarchy

- Cache coherence

Parallel Programming

Parallel software is the problem

Need to get significant performance improvement

- Otherwise, just use a faster uniprocessor (easier)
- Unless you can’t find a faster uniprocessor!

Difficulties

- Partitioning
- Coordination
- Communications overhead

Amdahl's Law

Sequential part can limit speedup

Example: 100 processors, 90× speedup?

$T_{new} = T_{parallelizable}/100 + T_{sequential}$

$$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$

Solving: $F_{parallelizable} = 0.999$

Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

What if matrix size is 100×100?

Single processor: Time = $(10 + 10000) \times t_{add}$

10 processors

- Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
- Speedup = 10010/1010 = 9.9 (99% of potential)

100 processors

- Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
- Speedup = 10010/110 = 91 (91% of potential)

Assuming load balanced

Strong vs Weak Scaling

Strong scaling: problem size fixed

- As in first example

Weak scaling: problem size proportional to number of processors

- 10 processors, 10 × 10 matrix
  - Time = $20 \times t_{add}$
- 100 processors, 32 × 32 matrix
  - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
- Constant performance in this example

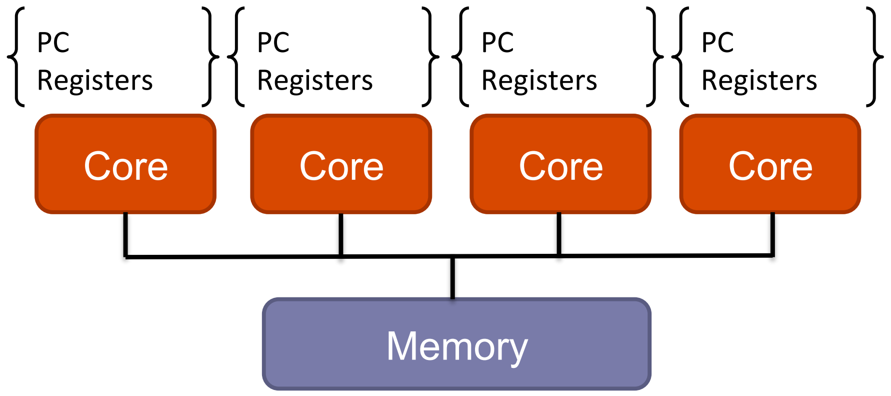
Flynn's Taxonomy: Classification of Parallel Machines

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel x86 processors	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Any multicore

SPMD: Single Program Multiple Data

- A parallel program on a MIMD computer
- Conditional code for different processors based on data

Shared Memory Multi-core



- Shared memory multi-core
  - Each core has private PC and registers
  - All cores access the same, shared memory
  - Communication through loads and stores
  - Synchronization through atomic instructions
- Requires cache coherence

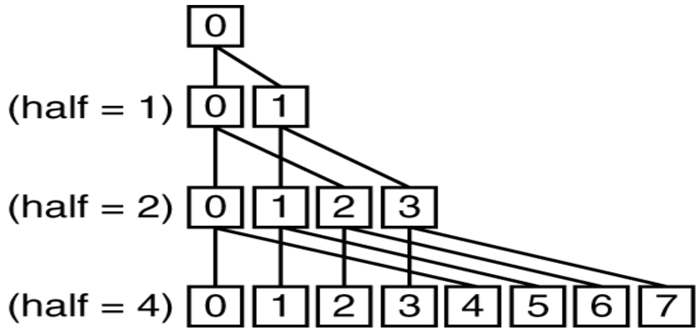
Example: Sum Reduction

- Sum 100,000 numbers on 100 processors
  - Each processor has ID: $0 \le Pn \le 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
      sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction



```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

Synchronization

- Assume multiple threads on multiple cores
- Shared data structure, e.g. a linked-list
- What happens when multiple threads work on the same list?
- How do we solve this issue?

Critical Sections (Locking)

```
bool void delete(item) {
        lock_list();
        found = lookup_item(item);
        if (found)
                delete_item(item);
        unlock_list();
}
```

What is the problem with this approach?

Solution? Fine grain locking

```
bool void delete(item) {
        found = lookup_item(item);
        if (found)
                lock_prev_item();
                delete_item(item);
                unlock_prev_item();
}
```

What do we need to implement locks?

```
bool lock = 0;
```

```
thread a:
if(lock==0)
lock = 1
//enter critical section
```

time

```
thread b:
if(lock==0)
lock = 1
//enter critical section
```

Atomic read-modify-write (RMW) instructions

Atomic Instructions in RISC-V

- load reserved (lr.w)
- store conditional (st.w)

- Example: (lock variable's memory address stored in x3)

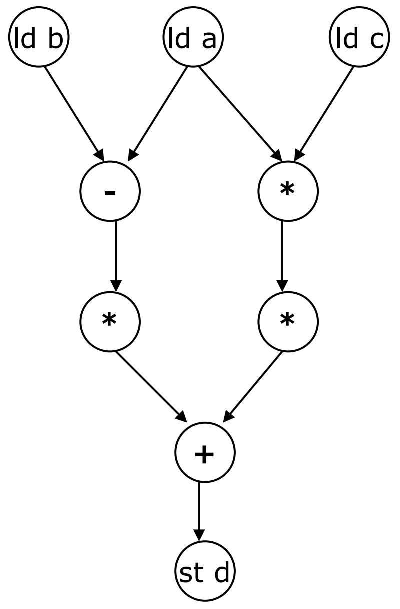
```
retry:
    lr.w x2, 0(x3)          //load lock variable into reg x2
    beq, x2, 1, retry       //check if already locked
    addi x2, x2, 1          //set lock variable in reg
    st.w x5, x2, 0(x3)      //store lock var, success: x5 <- 0
    bne x5, x0, retry       //check whether store was successful
```

Multithreading

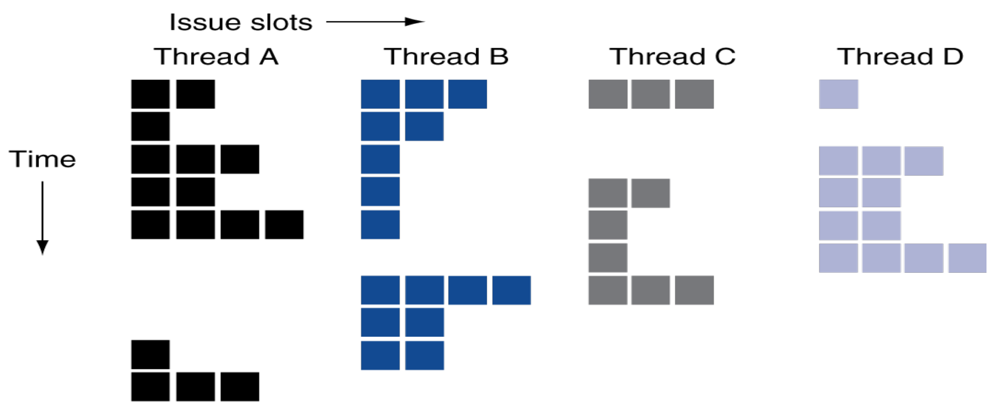
- Idea: execute multiple threads on a single core (or threads > cores)
- Threads within a process
  - Separate: PC, registers
  - Shared: virtual memory space
- Key advantage
  - When 1 thread stalls, switch to another thread
  - Stalls: pipeline, memory, I/O, …
  - The faster the switch, the more stall events we can mask

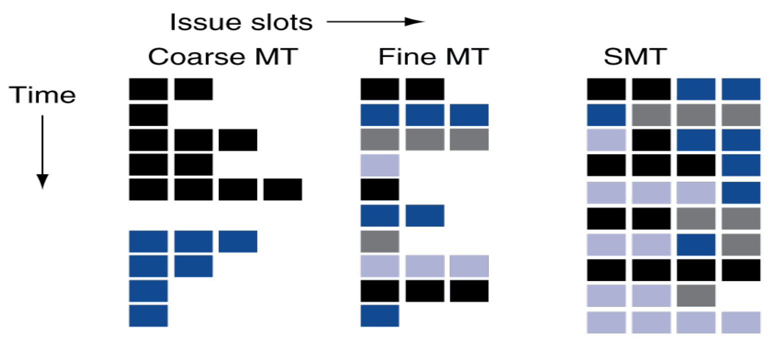
Reminder: Superscalar Processor

- Superscalar: Process >1 instruction per cycle
- Effectively multiple pipelines in parallel
- Exploits ILP by leveraging control flow graph
- Perform dynamic dependency checking



Types of Multithreading





Types of Multithreading

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed

- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (e.g., data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Options
  - Vectors/SIMD, multi-core, GPUs, clusters
- Important notes
  - SaaS importance is growing and clusters are a good match
  - Performance per $ and Joule drive both mobile and WSC