

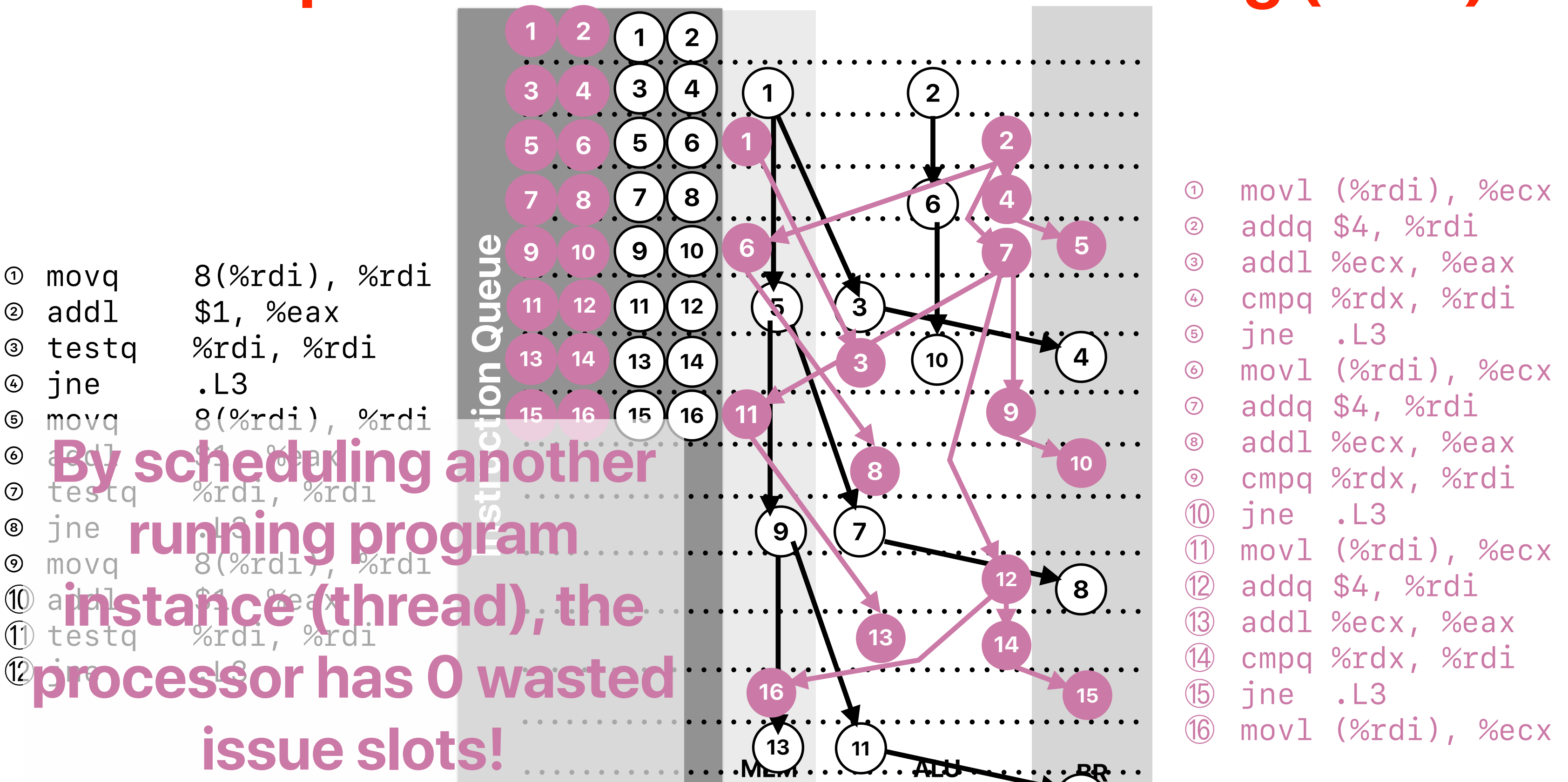
Programming on Multithreaded Architectures (2) & The Dark Silicon Problem

Hung-Wei Tseng

Recap: Parallelism in Modern Computers

- Instruction-level parallelism — concurrent execution of instructions from the same running program instance (process)
 - Superscalar
- Thread-level parallelism — concurrent execution of instructions from different running program instances
 - Simultaneous multithreading
 - Chip multiprocessor
- Data-level parallelism — concurrent execution of data streams from the same running program instance

Concept: Simultaneous Multithreading (SMT)





SMT from the user/OS' perspective




Recap: Transistor counts

Microarchitecture	Transistor Count	Issue-width	Year
Alder Lake	325 M	5x ALU, 7x Memory	2021
Coffee Lake	217 M	4x ALU, 4x Memory	2017
Sandy Bridge	290 M	3x ALU, 3x Memory	2011
Nehalem	182.75 M	3x ALU, 3x Memory	2008

 How many transistors per core on Coffee Lake?



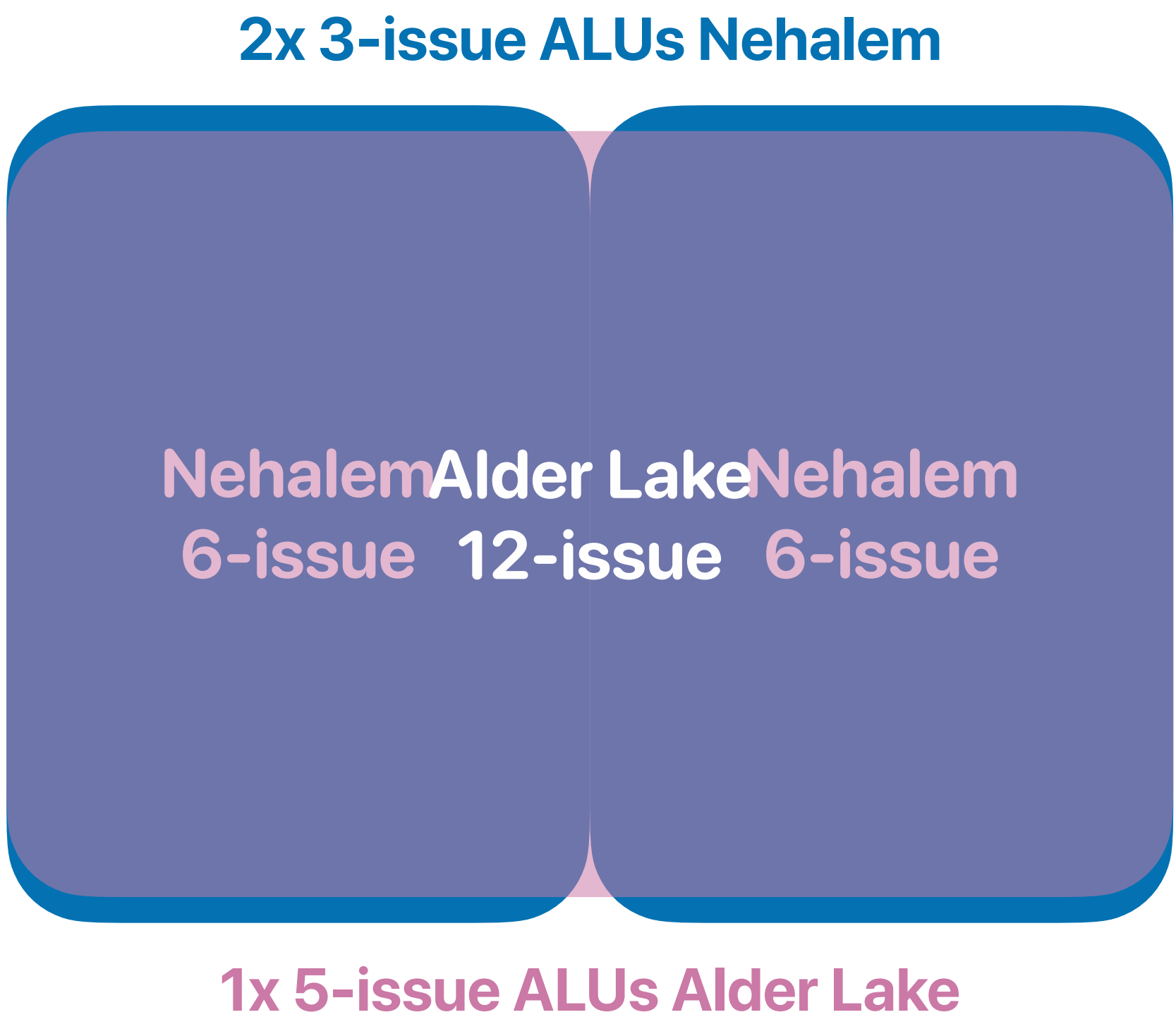
The Coffee Lake processor has 217 million transistors per core. It is manufactured using Intel's second 14 nm process. Coffee Lake processors introduced i5 and i7 CPUs featuring six cores (along with hyper-threading in the case of the i7) and no hyperthreading.



The transistor count per core on Coffee Lake is lower than that of some other modern processors, such as the AMD Ryzen 5 5600X. However, Coffee Lake still offers good performance, thanks to its high clock speeds and efficient architecture.

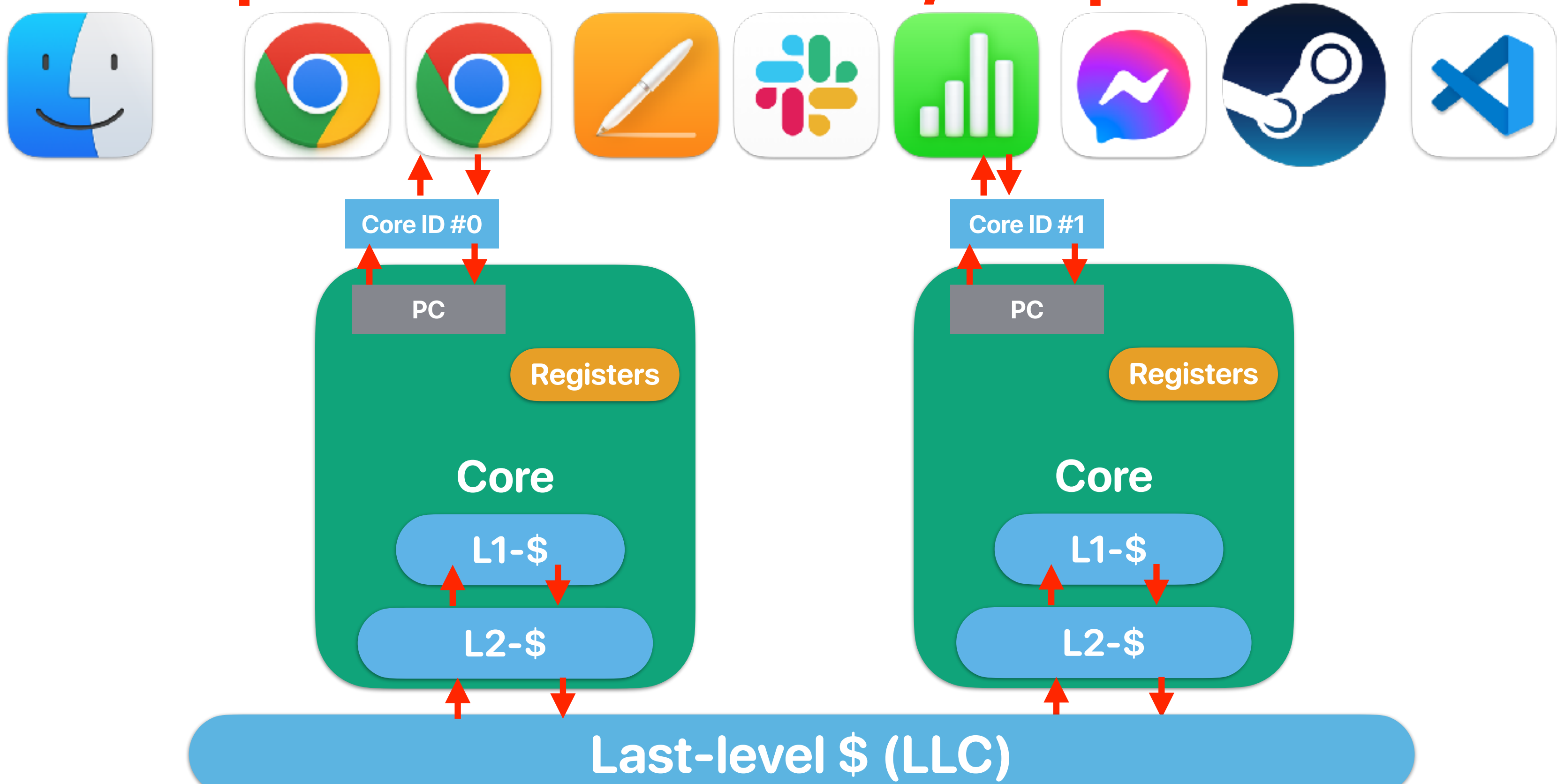
Here is a table of the transistor counts per core for some other modern processors:

Processor	Transistors per core
Coffee Lake	217 million
Ryzen 5 5600X	390 million
Ryzen 7 5800X	52 million



Based on https://en.wikipedia.org/wiki/Transistor_count

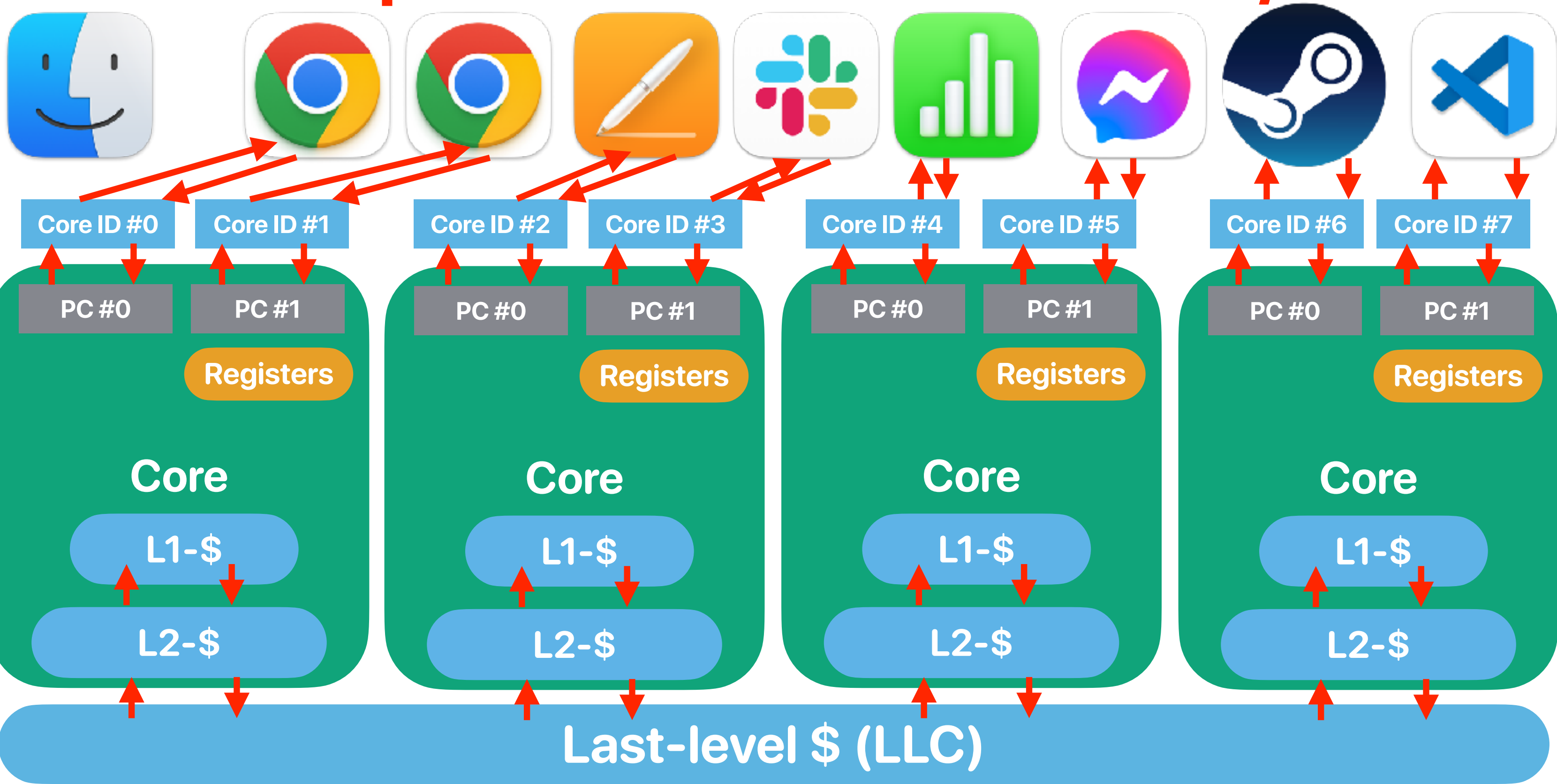
Recap: CMP from the user/OS' perspective



SMT v.s. CMP

- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many of the following statements are/is correct when running programs on these processors.
 - ① If we are just running one program in the system, the program will perform better on an SMT processor — **you have more resources for the program**
 - ② If we are running 4 applications simultaneously, the cache miss rates will be higher in the SMT processor
 - ③ If we are running 4 applications simultaneously, the branch mis-prediction will be higher in the SMT processor — **it depends!**
 - ④ If we are running one program with 4 parallel threads, the cache miss rates will be higher in the SMT processor — **it depends!**
 - ⑤ If we are running one program with 4 parallel threads simultaneously, the branch mis-prediction will be longer in the SMT processor — **it depends!**
- A. 1 **There is no clear win on each —**
B. 2 **why not having both?**
C. 3
D. 4 **The only thing we know for sure**
E. 5 **— if we don't parallel the program, it won't get any faster on CMP**

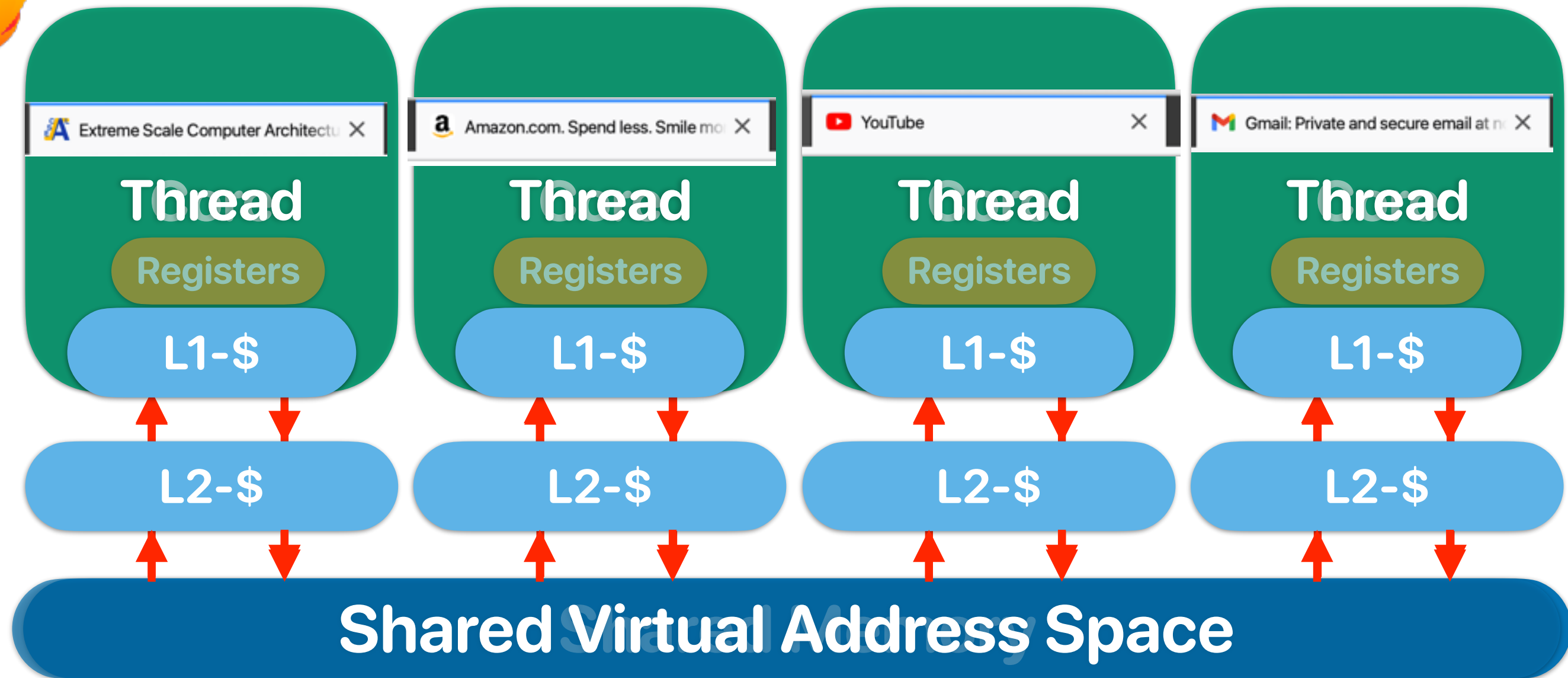
Modern processors have both CMP/SMT



Recap: Parallel programming

- To exploit parallelism you need to break your computation into multiple "processes" or multiple "threads"
- Processes (in OS/software systems)
 - Separate programs actually running (not sitting idle) on your computer at the same time.
 - Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs
- Threads (in OS/software systems)
 - Independent portions of your program that can run in parallel
 - All threads share the same virtual memory space
- We will refer to these collectively as "threads"
 - A typical user system might have 1-8 actively running threads.
 - Servers can have more if needed (the sysadmins will hopefully configure it that way)

Misaligned software/hardware abstraction



Coherency & Consistency

- Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
 - What value should be seen
- Consistency — All threads see the change of data in the same order
 - When the memory operation should be done

Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
<pre>while(1) printf("%d ", a);</pre>	<pre>while(1) a++;</pre>

- ① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100
A. 0
B. 1
C. 2
D. 3
E. 4

Outline

- Multithreaded processors
- Programming multithreaded processors & necessary architectural support

Parallel Programming & Architectural Supports for Parallel Programming (cont.)



Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid; i<ARRAY_SIZE; i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

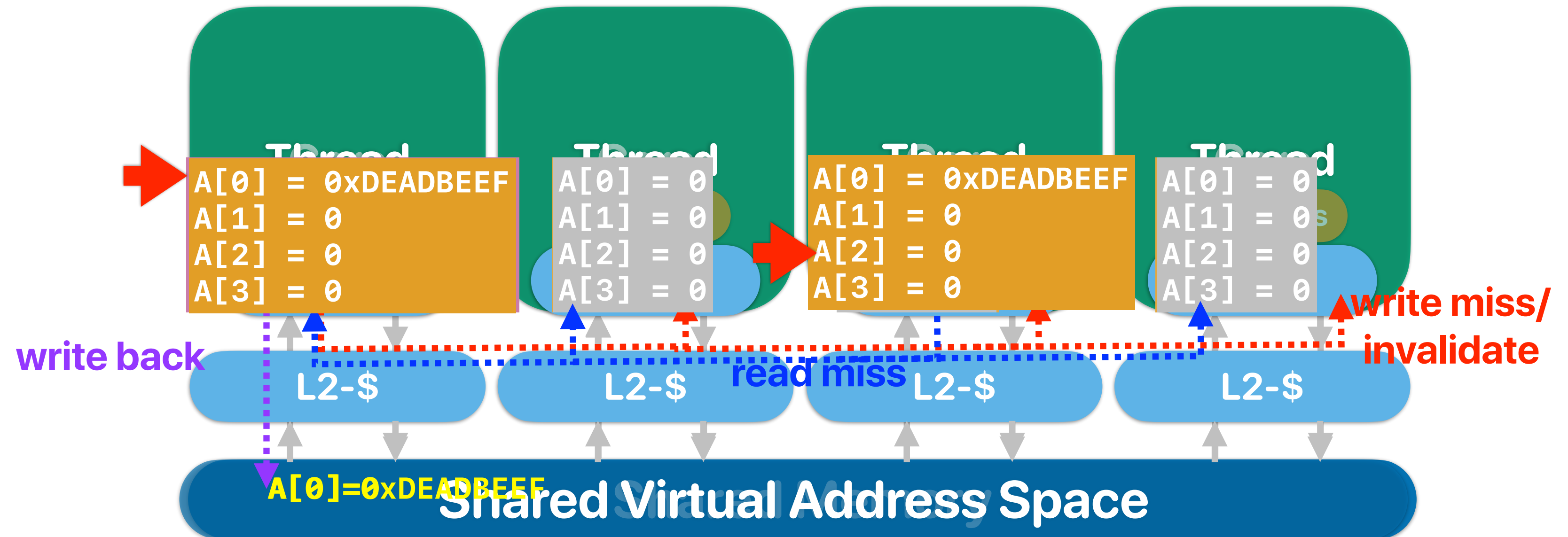
```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS); i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

- A. L is better, because the cache miss rate is lower
- B. R is better, because the cache miss rate is lower
- C. L is better, because the instruction count is lower
- D. R is better, because the instruction count is lower
- E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids[i])
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```


Cache coherency



Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid; i<ARRAY_SIZE; i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS); i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

- A. L is better, because the cache miss rate is lower
- B. R is better, because the cache miss rate is lower
- C. L is better, because the instruction count is lower
- D. R is better, because the instruction count is lower
- E. Both are about the same

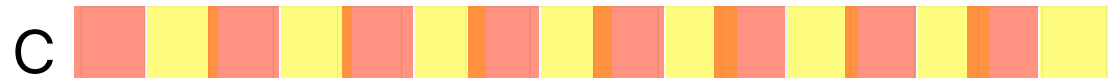
Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids[i])
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```

L v.s. R

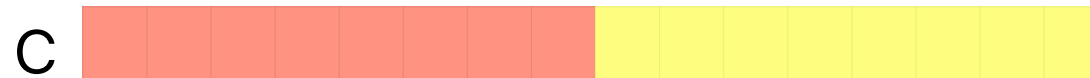
Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid; i<ARRAY_SIZE; i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```



Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS); i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```



4Cs of cache misses

- 3Cs:
 - Compulsory, Conflict, Capacity
- Coherency miss:
 - A "block" invalidated because of the sharing among processors.

False sharing

- True sharing
 - Processor A modifies X, processor B also want to access X.
- False sharing
 - Processor A modifies X, processor B also want to access Y.
However, Y is invalidated because X and Y are in the same block!

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid; i<ARRAY_SIZE; i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS); i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

- A. L is better, because the cache miss rate is lower
- B. R is better, because the cache miss rate is lower**
- C. L is better, because the instruction count is lower
- D. R is better, because the instruction count is lower
- E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids[i])
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```



Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)

② (0, 1)

③ (1, 0)

④ (1, 1)

A. 0

B. 1

C. 2

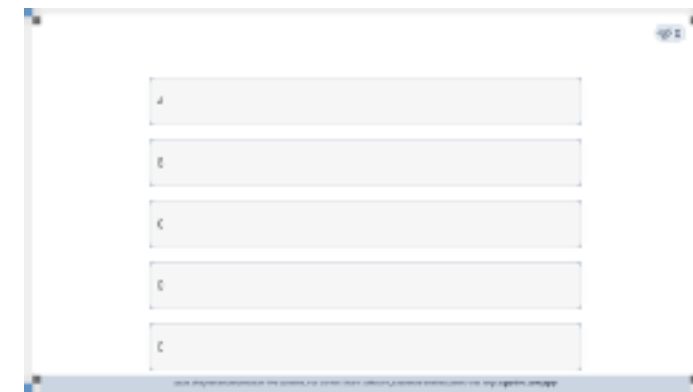
D. 3

E. 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
    a=1;
    x=b;
    return NULL;
}
void* modifyb(void *z) {
    b=1;
    y=a;
    return NULL;
}
```

```
int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n", x, y);
    return 0;
}
```



Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)

② (0, 1)

③ (1, 0)

④ (1, 1)

A. 0

B. 1

C. 2

D. 3

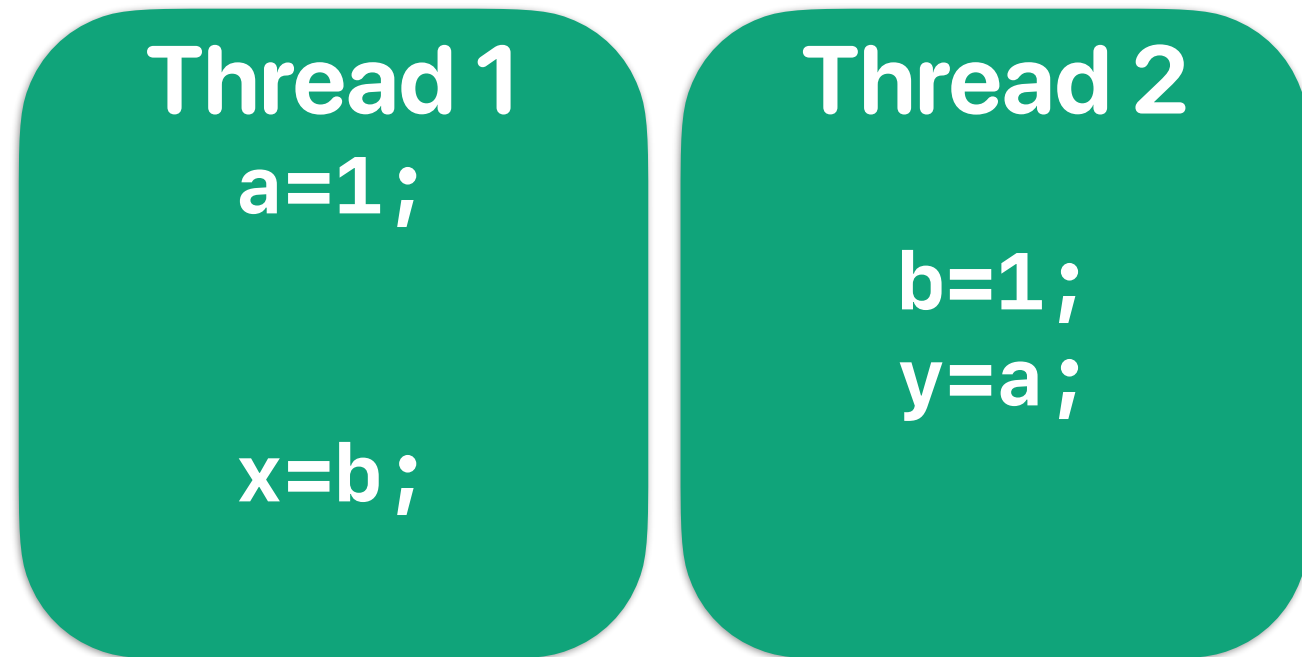
E. 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

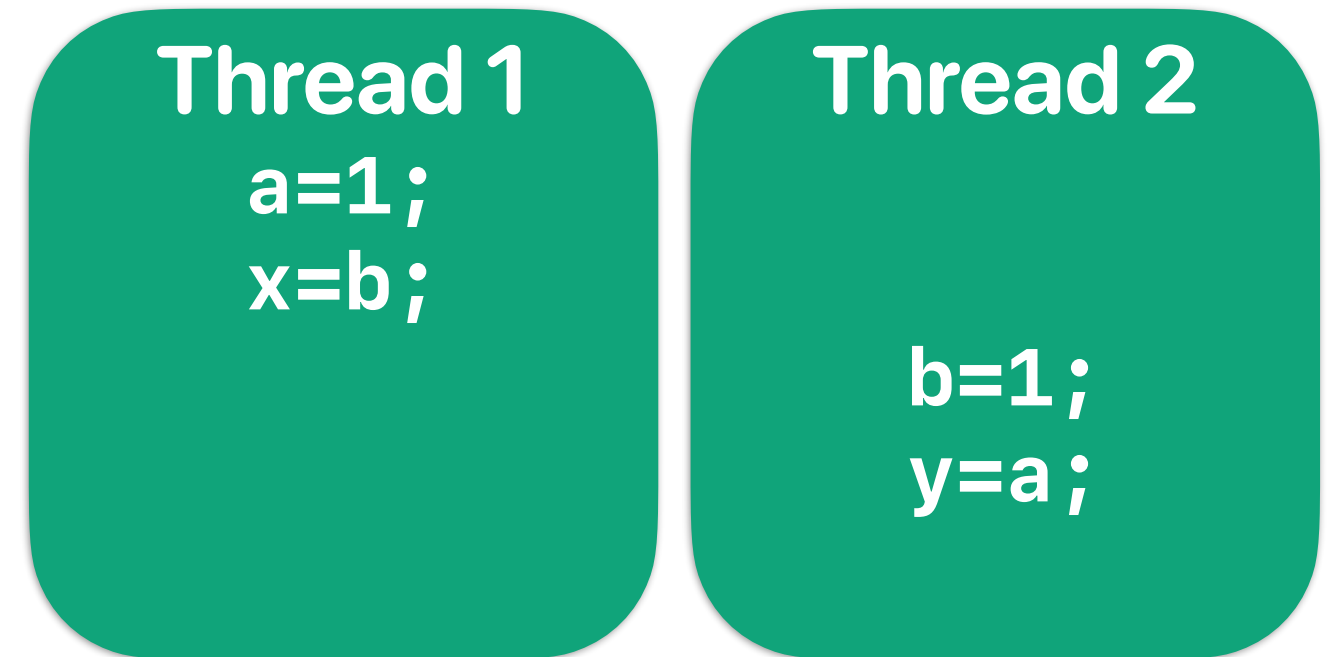
volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
    a=1;
    x=b;
    return NULL;
}
void* modifyb(void *z) {
    b=1;
    y=a;
    return NULL;
}
```

```
int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n", x, y);
    return 0;
}
```

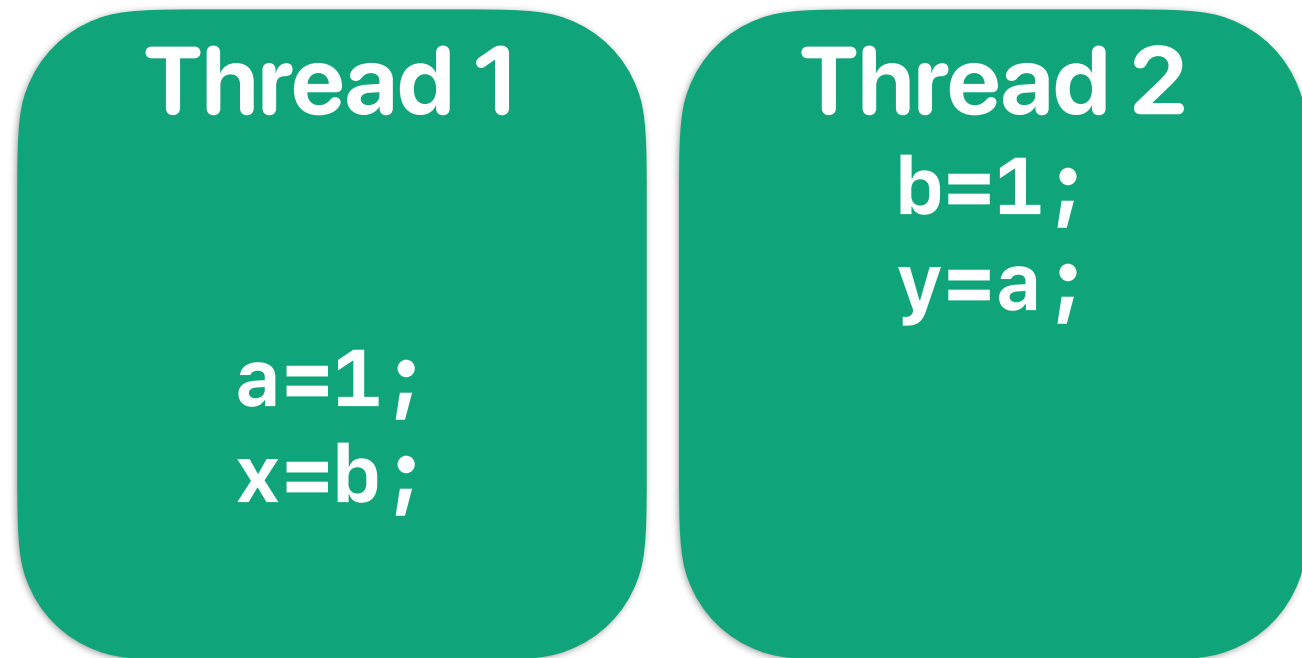
Possible scenarios



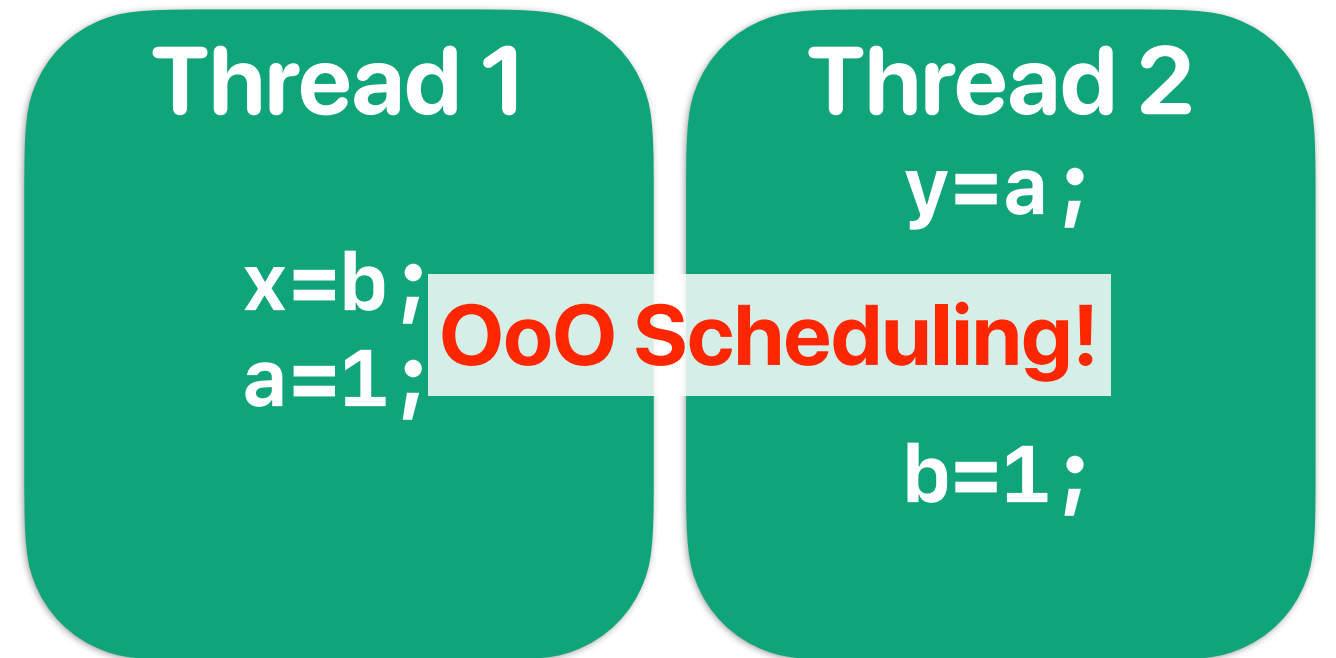
(1,1)



(0,1)



(1,0)



(0,0)

Why (0,0)?

- Processor/compiler may reorder your memory operations/instructions
 - Coherence protocol can only guarantee the update of the same memory address
 - Processor can serve memory requests without cache miss first
 - Compiler may store values in registers and perform memory operations later
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it's spawned

Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)

② (0, 1)

③ (1, 0)

④ (1, 1)

A. 0

B. 1

C. 2

D. 3

E. 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
    a=1;
    x=b;
    return NULL;
}
void* modifyb(void *z) {
    b=1;
    y=a;
    return NULL;
}
```

```
int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n", x, y);
    return 0;
}
```

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
 - All updates prior to mfence must finish before the instruction can proceed
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

thread 1	thread 2
<pre>a=1; mfence x=b;</pre> <p>a=1 must occur/update before mfence</p>	<pre>b=1; mfence y=a;</pre> <p>b=1 must occur/update before mfence</p>

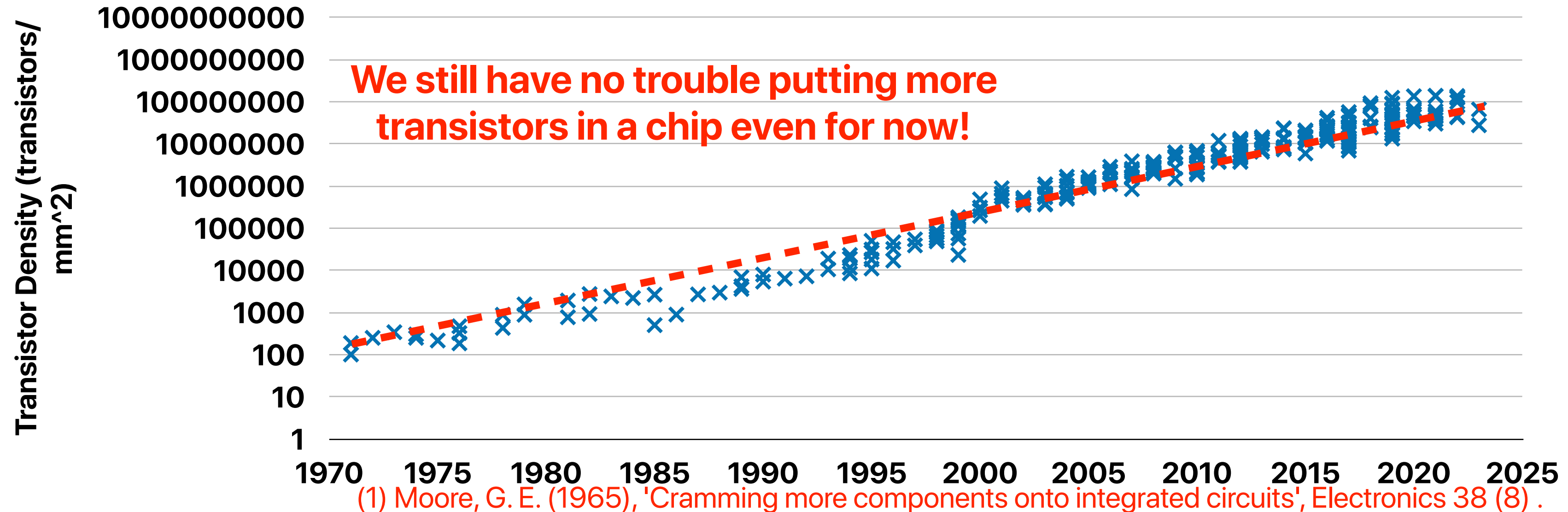
Take-aways of parallel programming

- Processor behaviors are non-deterministic
 - You cannot predict which processor is going faster
 - You cannot predict when OS is going to schedule your thread
- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache consistency is hard to support

The limiting factor of modern processor performance

Moore's Law⁽¹⁾

- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains





What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?
 - ① The power consumption per chip will increase
 - ② The power density of the chip will increase
 - ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
 - ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0
B. 1
C. 2
D. 3
E. 4

A screenshot of a web-based poll interface. It displays five empty, light-gray rectangular input boxes stacked vertically, intended for users to enter their answers to the question.

Power v.s. Energy

- Power is the direct contributor of "heat"
 - Packaging of the chip
 - Heat dissipation cost
- $\text{Energy} = P * ET$
 - The electricity bill and battery life is related to energy!
 - Lower power does not necessary means better battery life if the processor slow down the application too much

Dynamic/Active Power

- The power consumption due to the switching of transistor states

- Dynamic power per transistor

$$P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$$

- α : average switches per cycle
- C : capacitance
- V : voltage
- f : frequency, usually linear with V
- N : the number of transistors

Static/Leakage Power

- The power consumption due to leakage — transistors do not turn all the way off during no operation
- Becomes the **dominant** factor in the most advanced process technologies.

$$P_{leakage} \sim N \times V \times e^{-V_t}$$

- N : number of transistors
- V : voltage
- V_t : threshold voltage where transistor conducts (begins to switch)

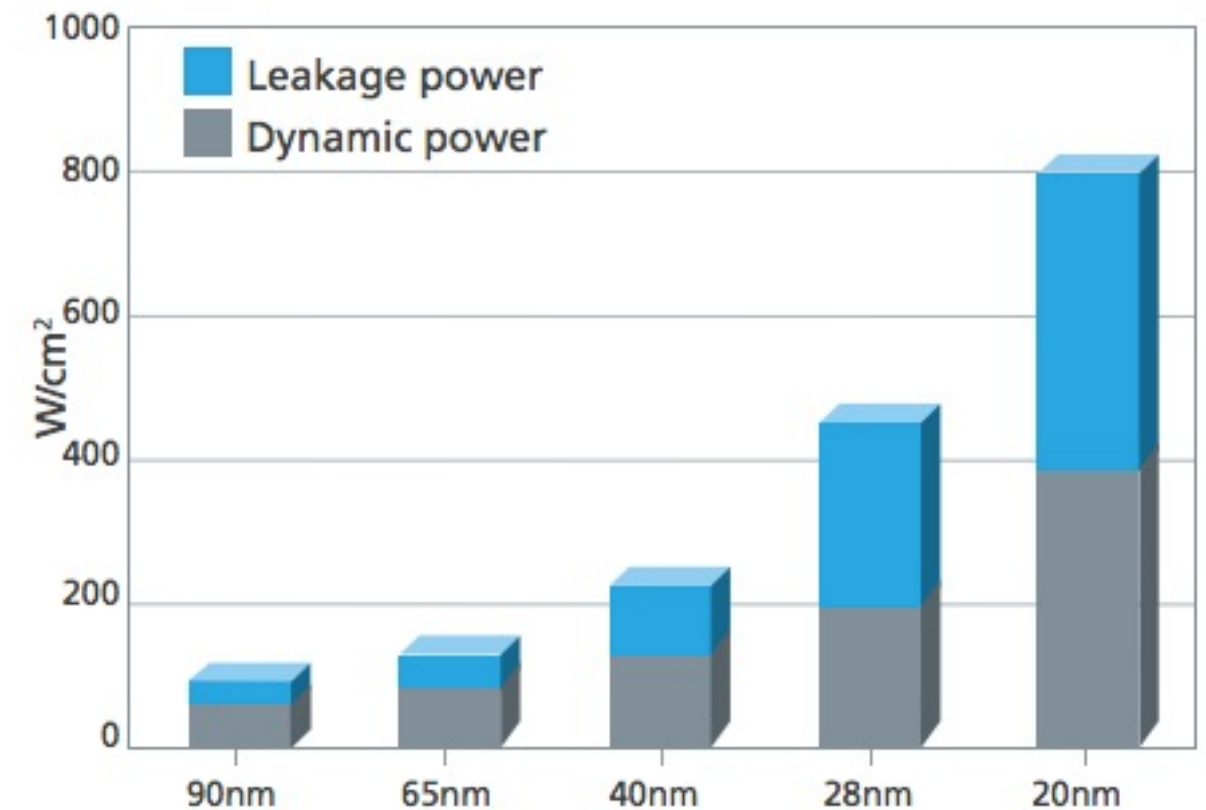


Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).

Power consumption & power density

- $P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$

- α : average switches per cycle

- C : capacitance

- V : voltage

- f : frequency, usually linear with V

- N : the number of transistors

- $P_{leakage} \sim N \times V \times e^{-V/V_t}$

- N : number of transistors

- V : voltage

- V_t : threshold voltage where transistor conducts (begins to switch)

- Power density:

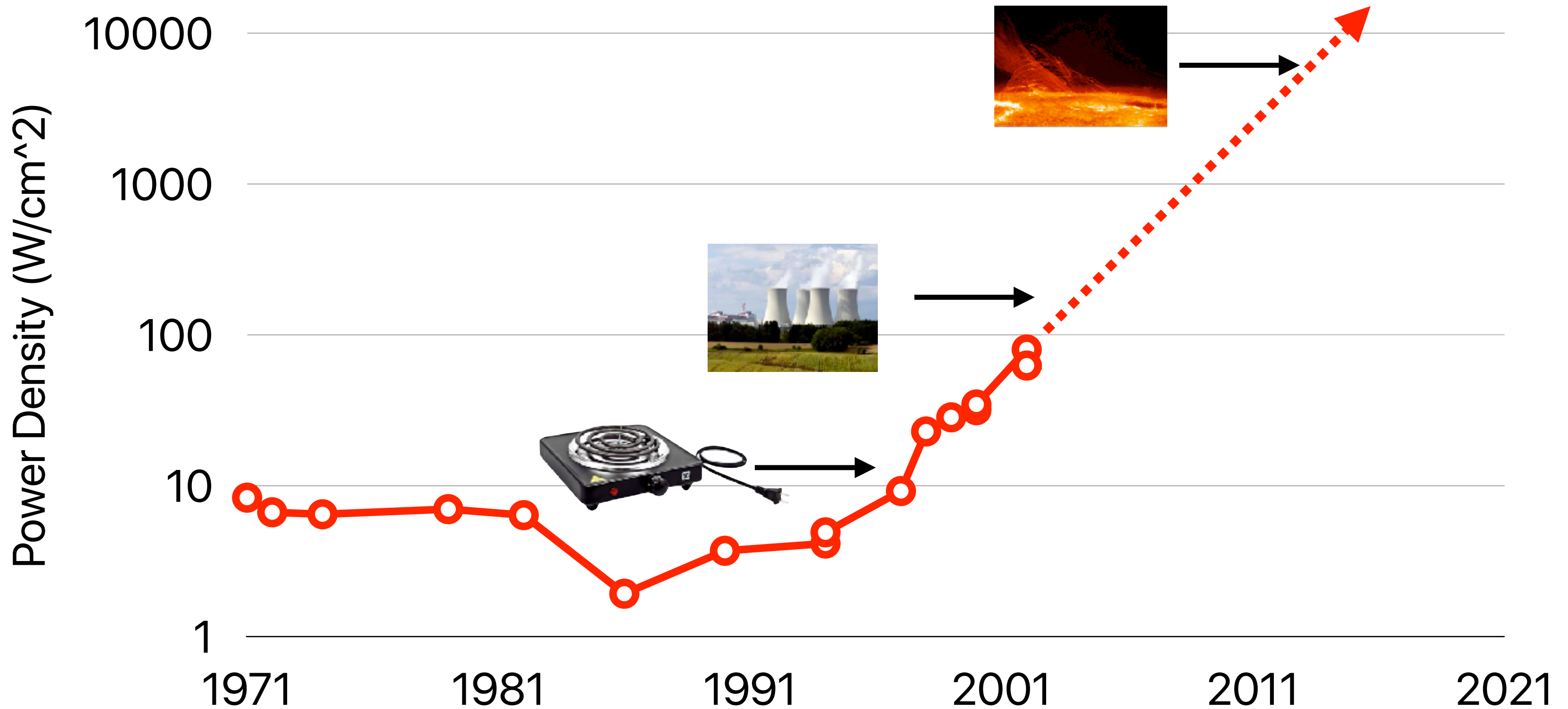
$$P_{density} = \frac{P}{area}$$

Dennard scaling discontinued — we cannot make voltage lower

Moore's Law allows higher frequencies as transistors are smaller

Moore's Law makes this smaller

Power Density of Processors



Power consumption to light on all transistors

Dennardian Scaling

Dennardian Broken

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Chip

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

On ~
50W

Off ~
0W

Dark!

=100W!

Or we have to dim down all transistors

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Dennardian Scaling

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Dennardian Broken

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

**Low frequency
~ 0.5W**

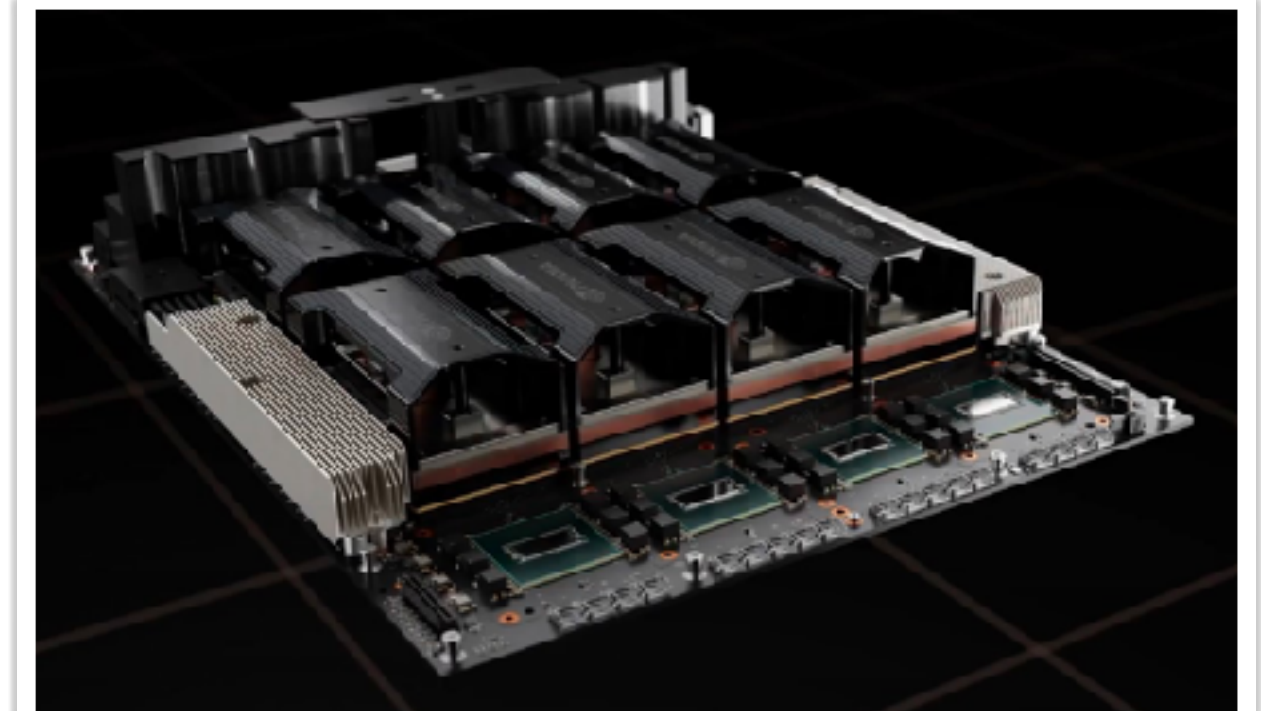
=50W!

If you can add power budget...

NVIDIA Accelerator Specification Comparison			
	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)
GPU	GH100 (814mm ²)	GA100 (826mm ²)	GV100 (815mm ²)
Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FFN
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta



<https://www.workstationspecialist.com/product/nvidia-tesla-a100/>



<https://www.servethehome.com/wp-content/uploads/2022/03/NVIDIA-GTC-2022-H100-in-HGX-H100.jpg>

What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?

- ① The power consumption per chip will increase
- ② The power density of the chip will increase
- ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
- ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0

B. 1

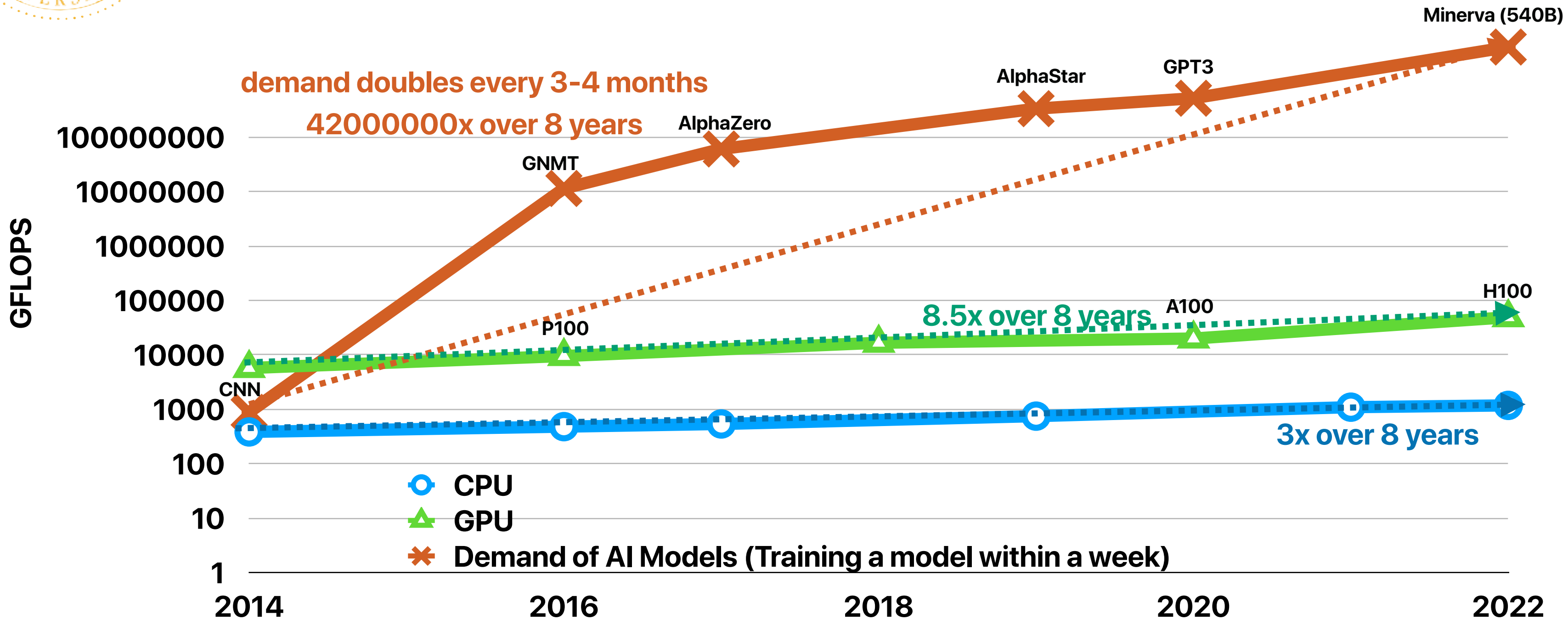
C. 2

D. 3

E. 4



Mis-matching AI/ML demand and general-purpose processing



<https://ourworldindata.org/grapher/artificial-intelligence-training-computation>

Announcements

- Assignment #5 due next Tuesday
- iEVAL — if you fill out iEVAL, please submit a screenshot
 - You will receive a full credit reading quiz if you submit the screenshot before the in-person final examine
 - We will drop your **lowest two** reading quizzes — so that if you don't want to fill the form, you still have the lowest one dropped
- Final Exam — will release some sample questions next Tuesday at the end of the lecture
 - 12/7 3:30p—4:50p
 - Including CSMS comprehensive examine questions — you need to answer them even you're not a CS MS.
 - In-person, no outside materials
 - 12/11-12/14 6pm—6pm — any consecutive three-hour slot you pick
 - Online examine
 - Including coding assessments (similar to company interviews)
 - Essay-style questions that require thorough understandings of course materials

Computer Science & Engineering

203

つづく

