

Lecture 5

# Processor Architecture & Efficiency

How to write efficient numerical code for today's hardware

## CS328 - Numerical Methods for Visual Computing and Machine Learning

Prof. Wenzel Jakob

# Course evaluations

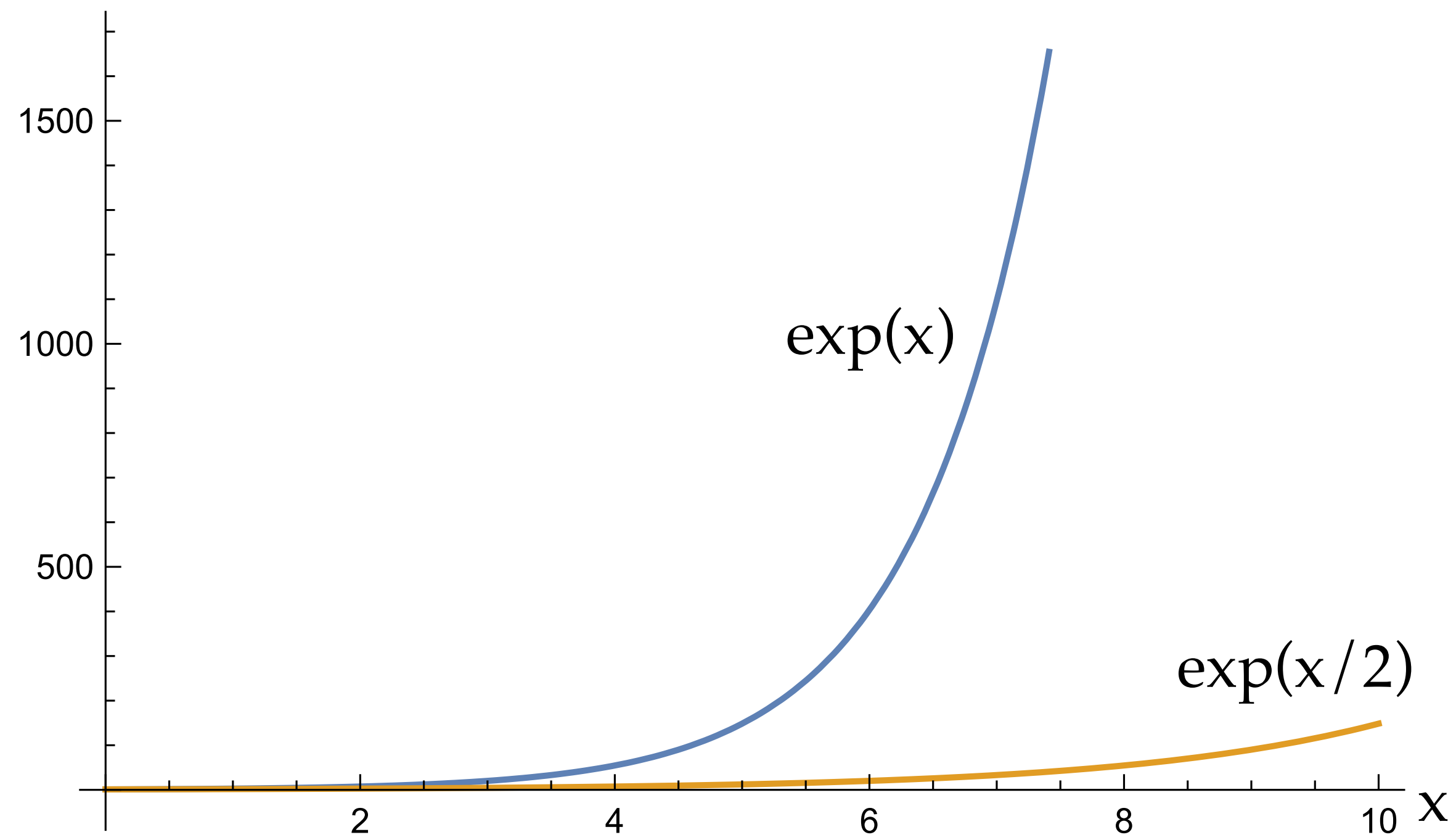
Don't forget to review your courses (including CS328) until October 22

- What you write in there **really matters**.
  - I read it. The TAs read it.
  - The IC sections read it.
  - If it's very bad (or *very good*) it, the IC sections follows up with course instructors.
  - Curious about your thoughts!

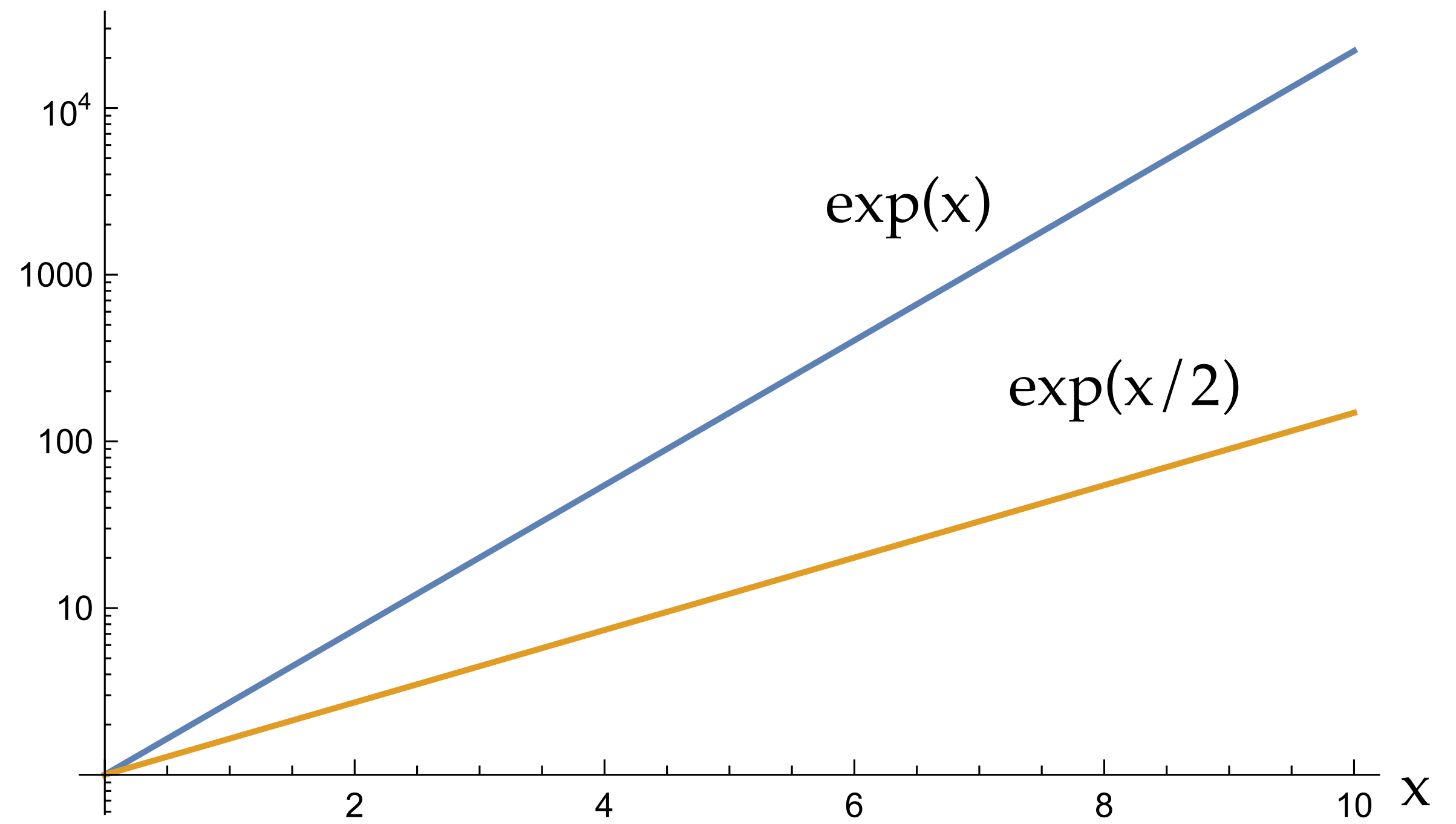


**MidJourney:** a person putting a piece of paper into a ballot box, on white background.

# A Semi-Log Plot: What is it? How to read it?

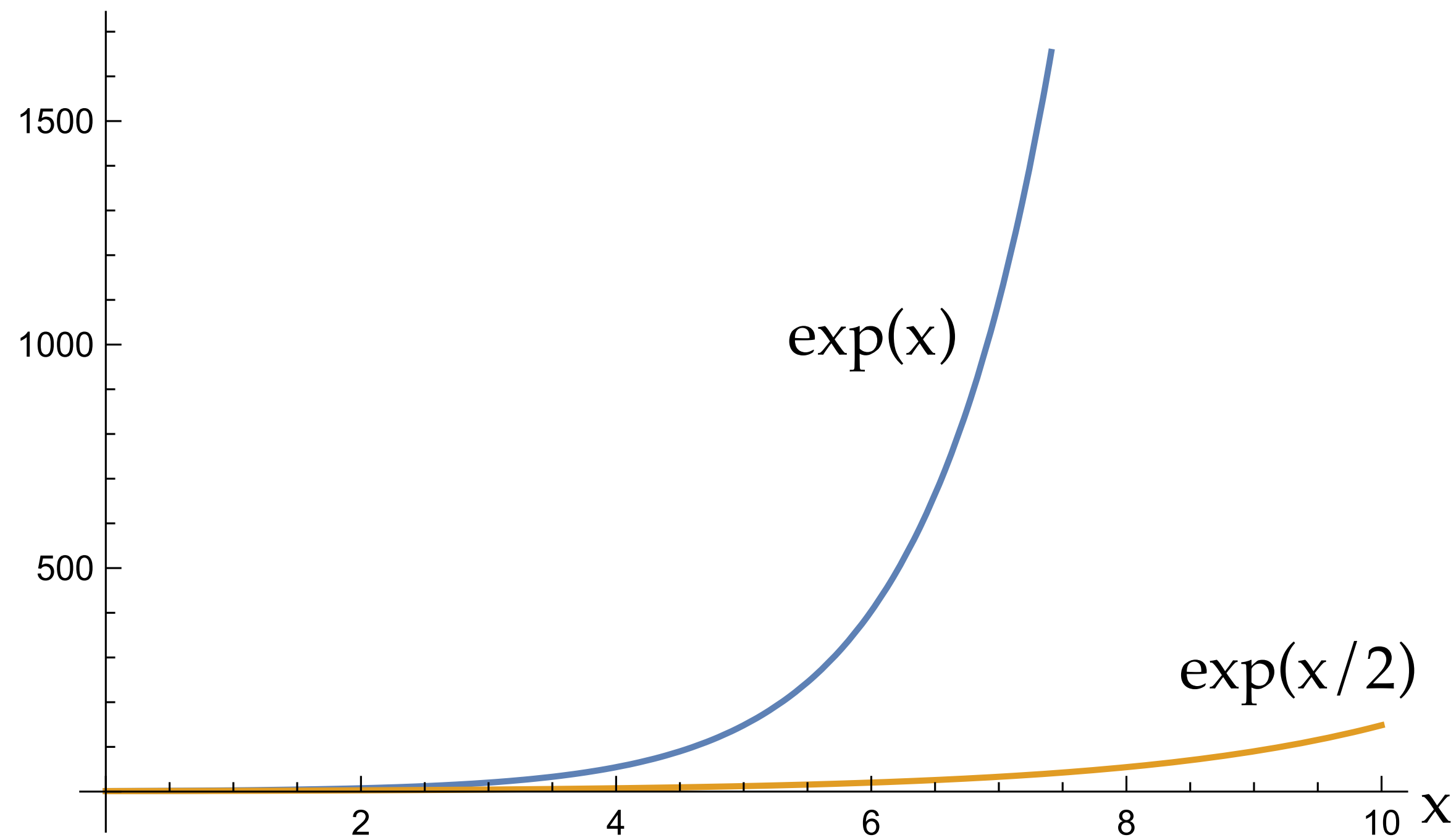


Linear plot

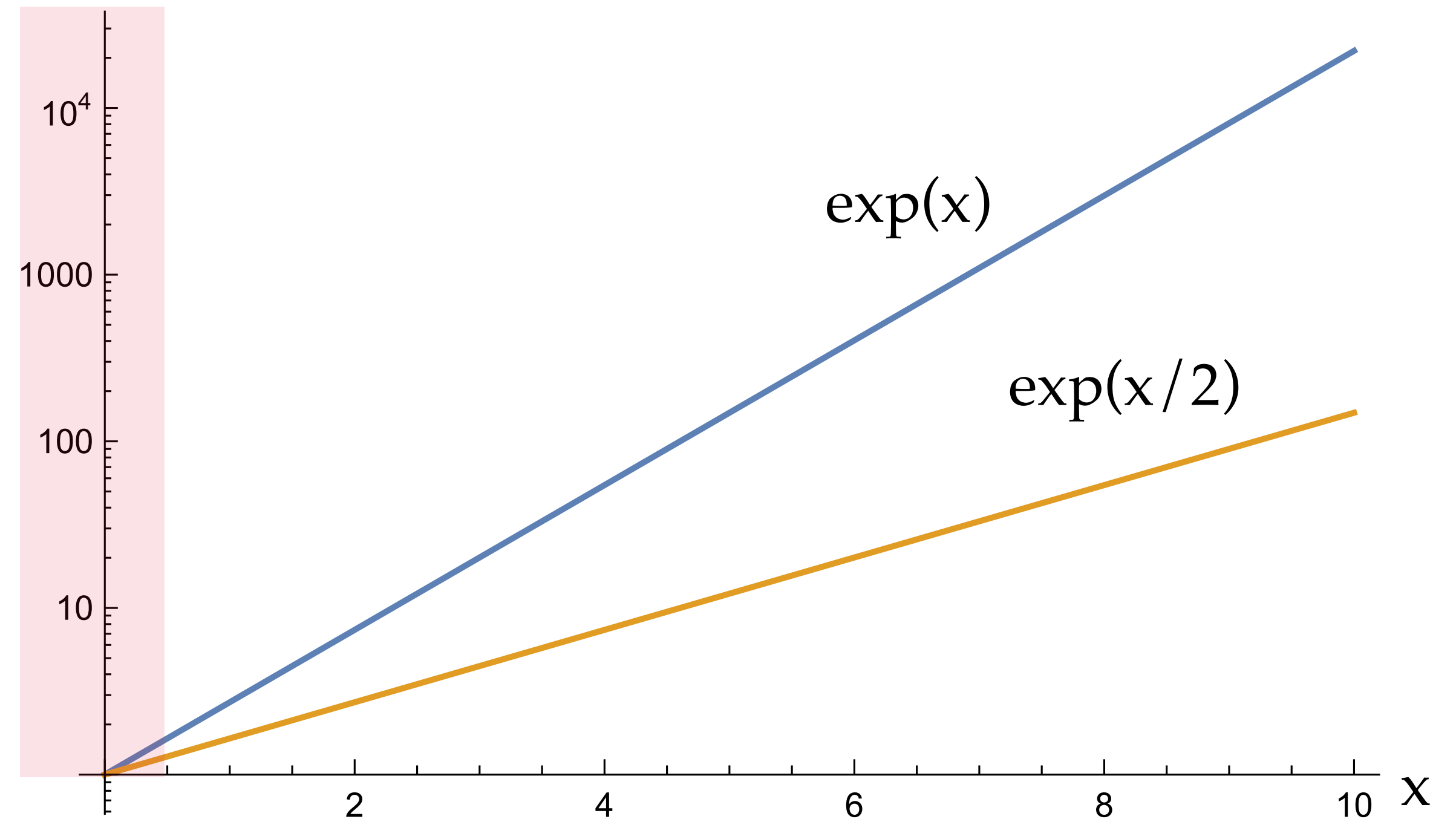


Semi-log plot

# A Semi-Log Plot: What is it? How to read it?



Linear plot

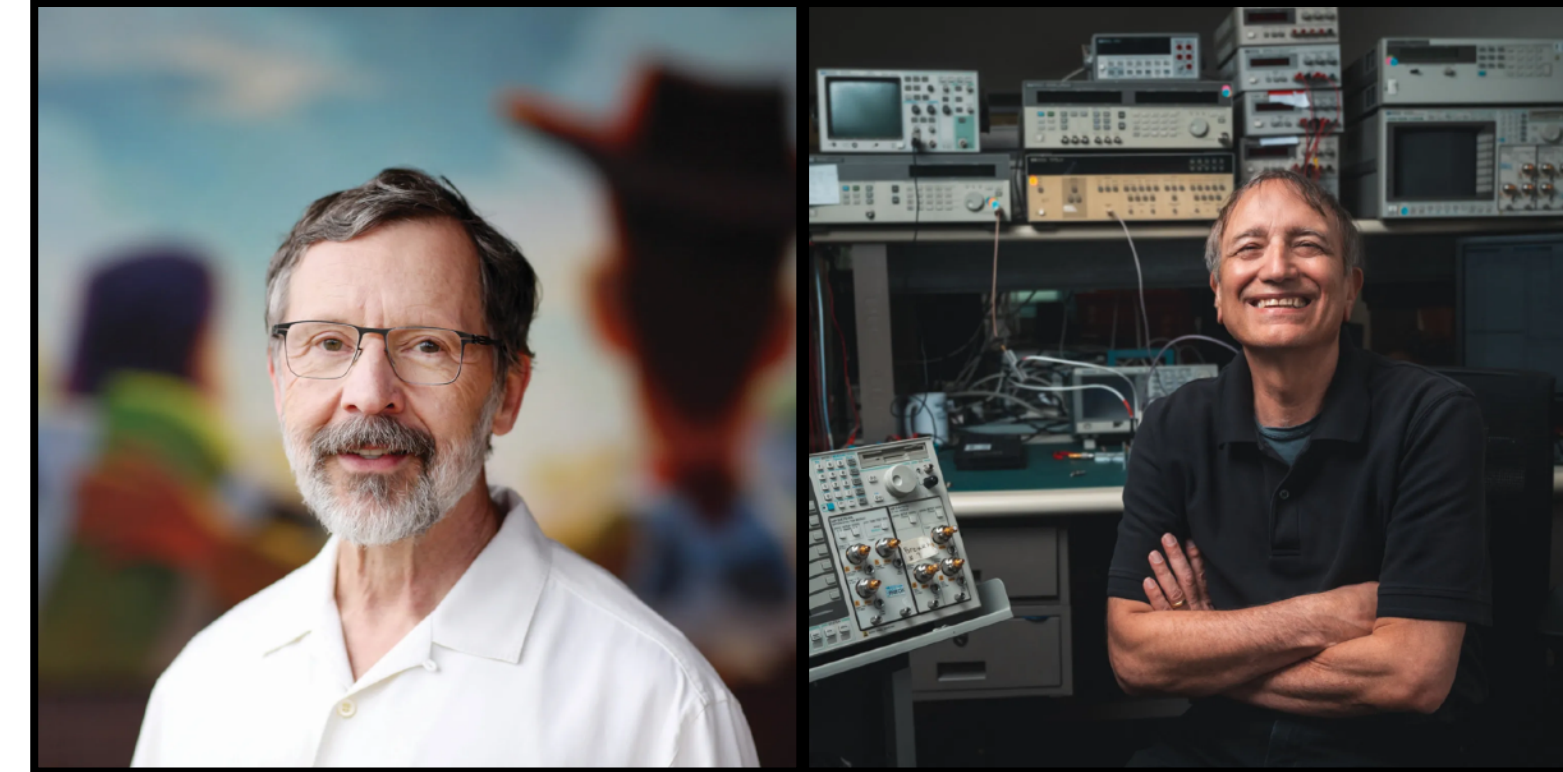


Semi-log plot

# Motivation: A Story of Nonlinear Growth

- **2020 ACM Turing Award**  
"Nobel Prize" for Computer Science

Image Credits: Deborah Coleman / Pixar; Andrew Brodhead / Stanford University



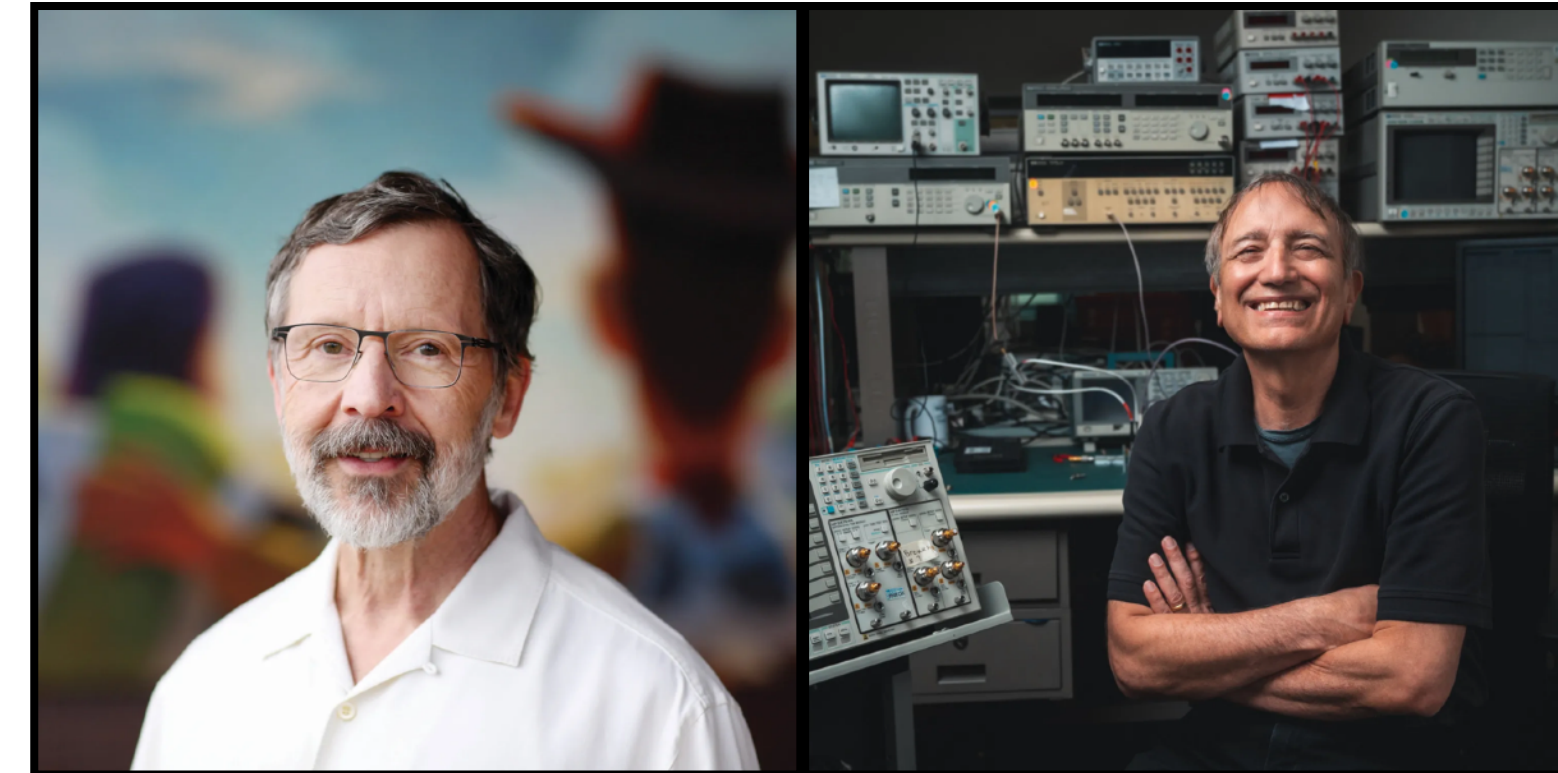
Ed Catmull

Pat Hanrahan

# Motivation: A Story of Nonlinear Growth

- **2020 ACM Turing Award**  
"Nobel Prize" for Computer Science
- 1979: Catmull joined Lucasfilm to create a *computer division* and make a digital movie.

Image Credits: Deborah Coleman / Pixar; Andrew Brodhead / Stanford University



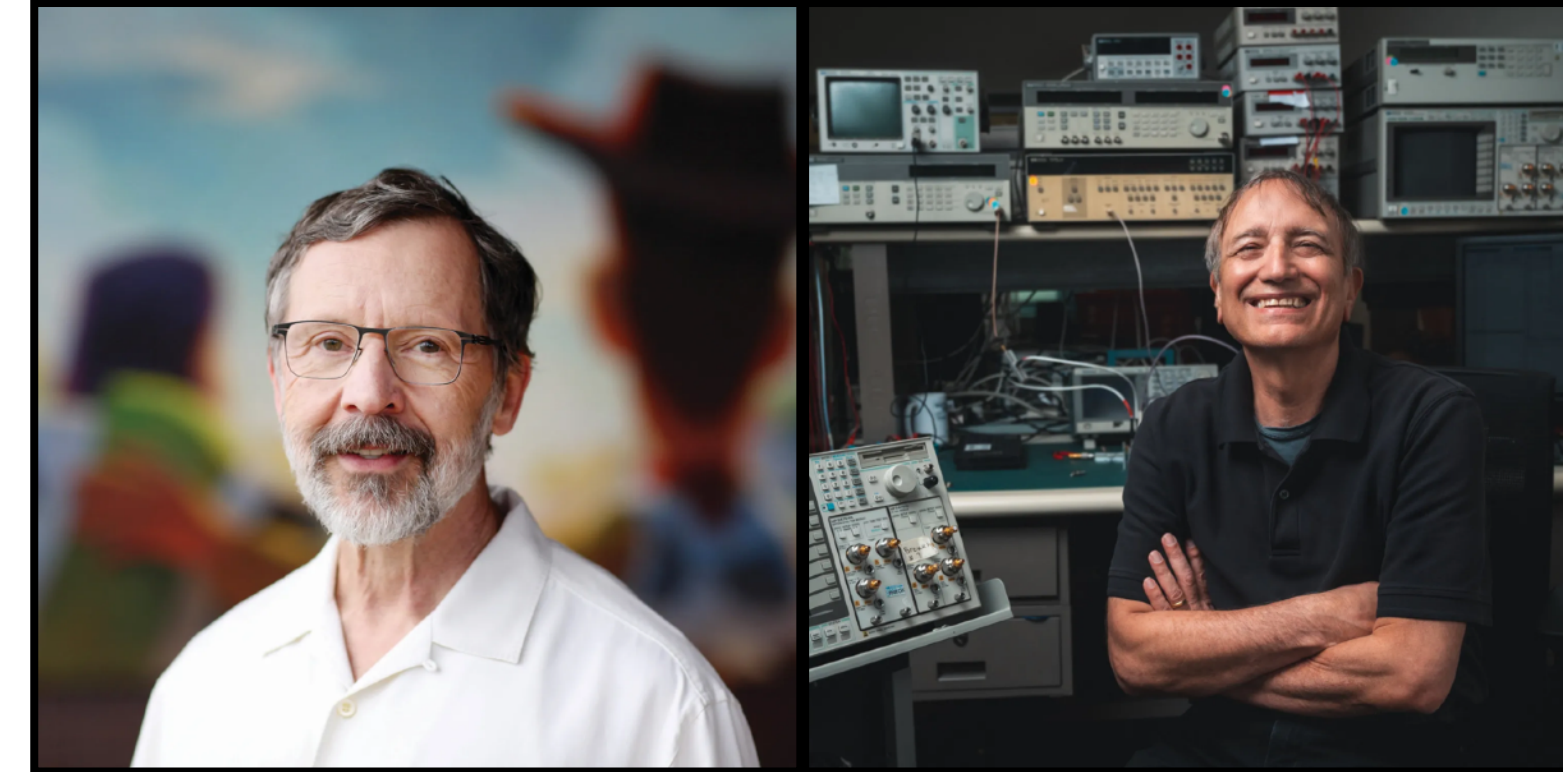
Ed Catmull

Pat Hanrahan

# Motivation: A Story of Nonlinear Growth

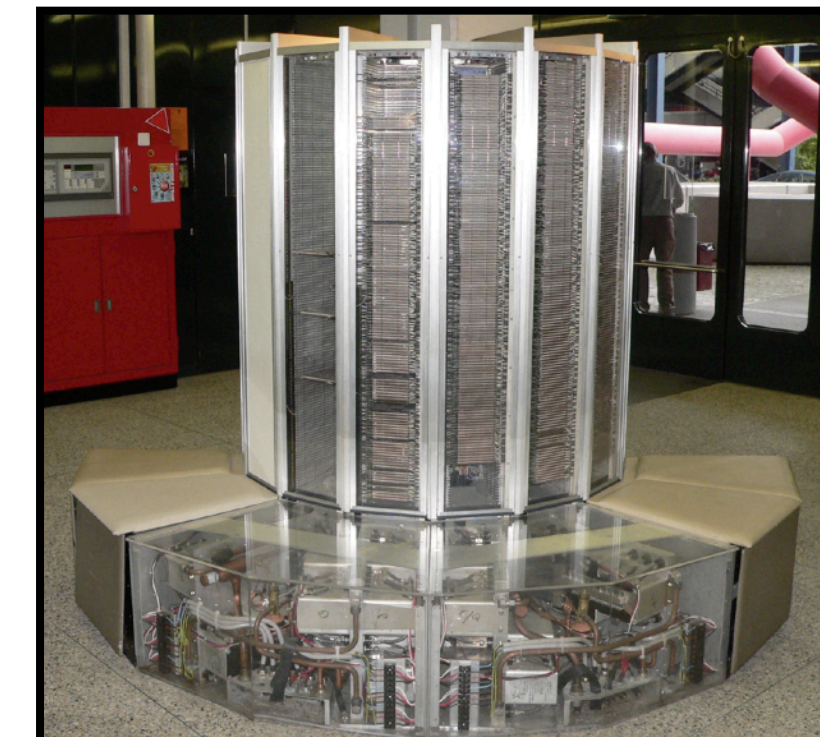
- **2020 ACM Turing Award**  
"Nobel Prize" for Computer Science
- 1979: Catmull joined Lucasfilm to create a *computer division* and make a digital movie.
- A **preposterous** idea: would have needed the power of 1000 CRAY-1 supercomputers (10 million US \$ for just one)...  
.. but only had a budget of 1M \$.

Image Credits: Deborah Coleman / Pixar; Andrew Brodhead / Stanford University



Ed Catmull

Pat Hanrahan

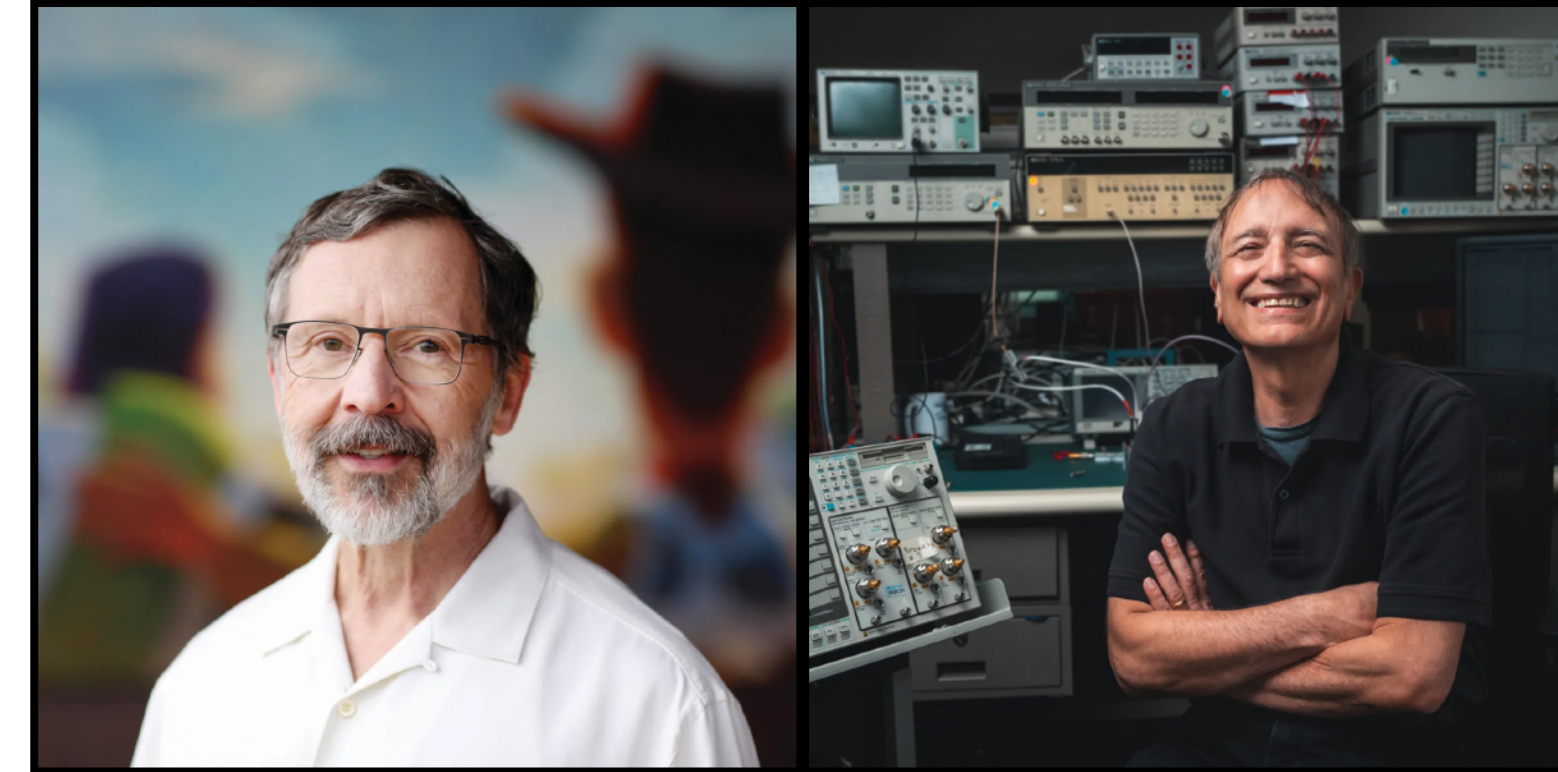


CRAY-1  
Supercomputer

# Motivation: A Story of Nonlinear Growth

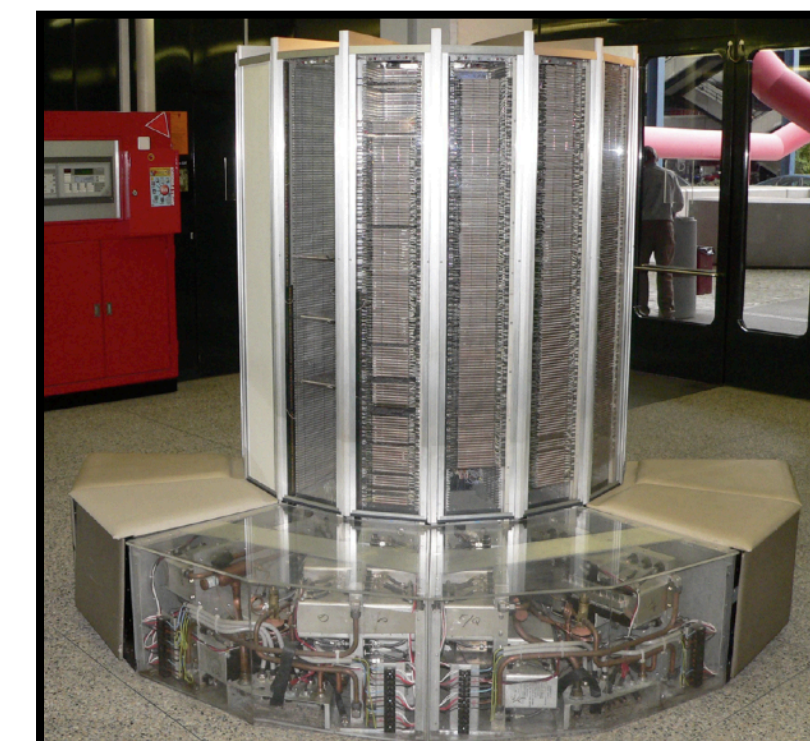
- **2020 ACM Turing Award**  
"Nobel Prize" for Computer Science
- 1979: Catmull joined Lucasfilm to create a *computer division* and make a digital movie.
- A **preposterous** idea: would have needed the power of 1000 CRAY-1 supercomputers (10 million US \$ for just one)...  
.. but only had a budget of 1M \$.
- Catmull: "*Based on where we were on the exponential curve for computing speed, it would take another 14-15 years. Our time and resources were better spent on the many problems we could see here and now. Coincidentally, when we released "Toy Story" about 15 years later, we were very close to our estimate of the computing power required.*"

Image Credits: Deborah Coleman / Pixar; Andrew Brodhead / Stanford University



Ed Catmull

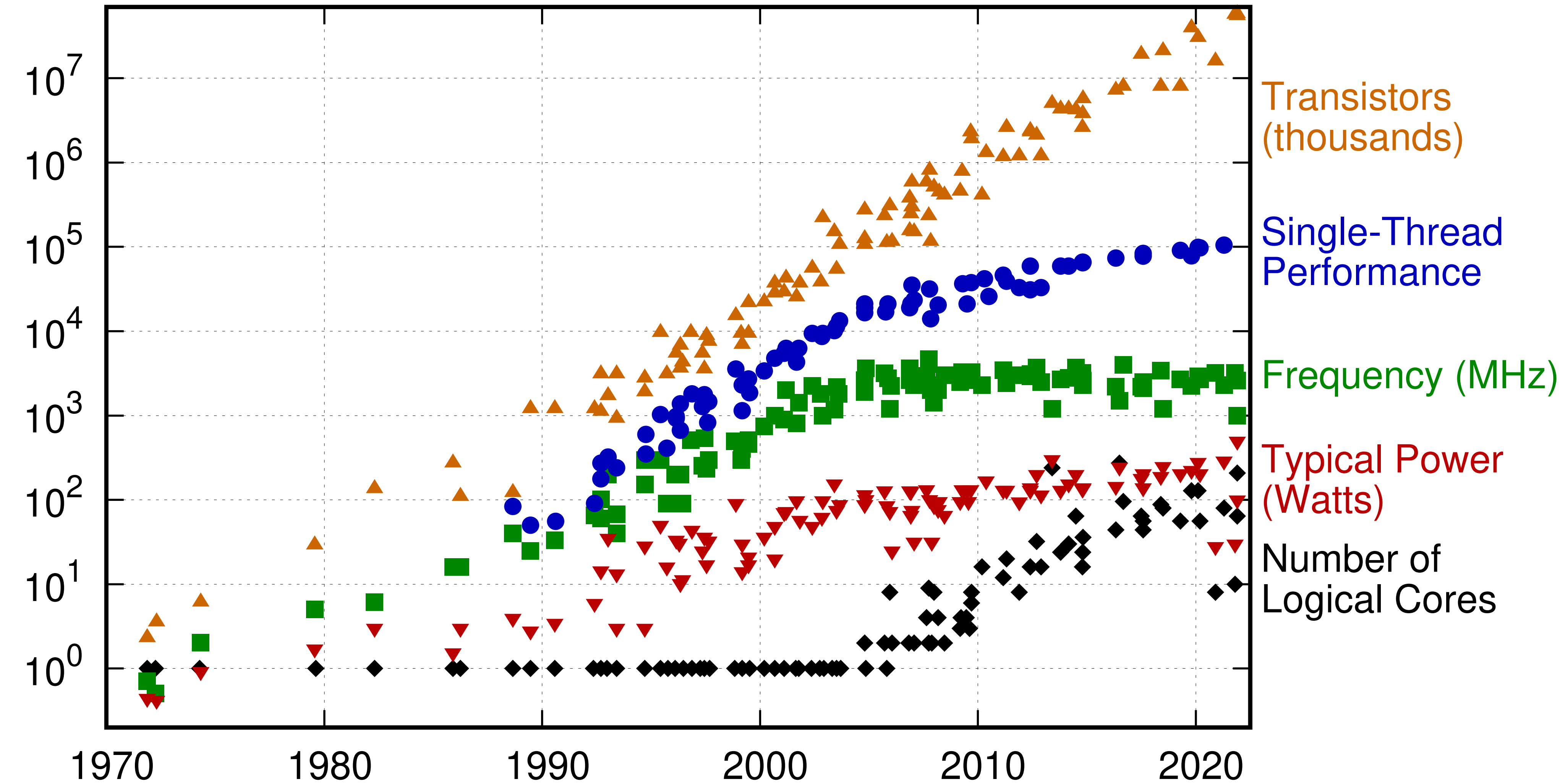
Pat Hanrahan



CRAY-1  
Supercomputer

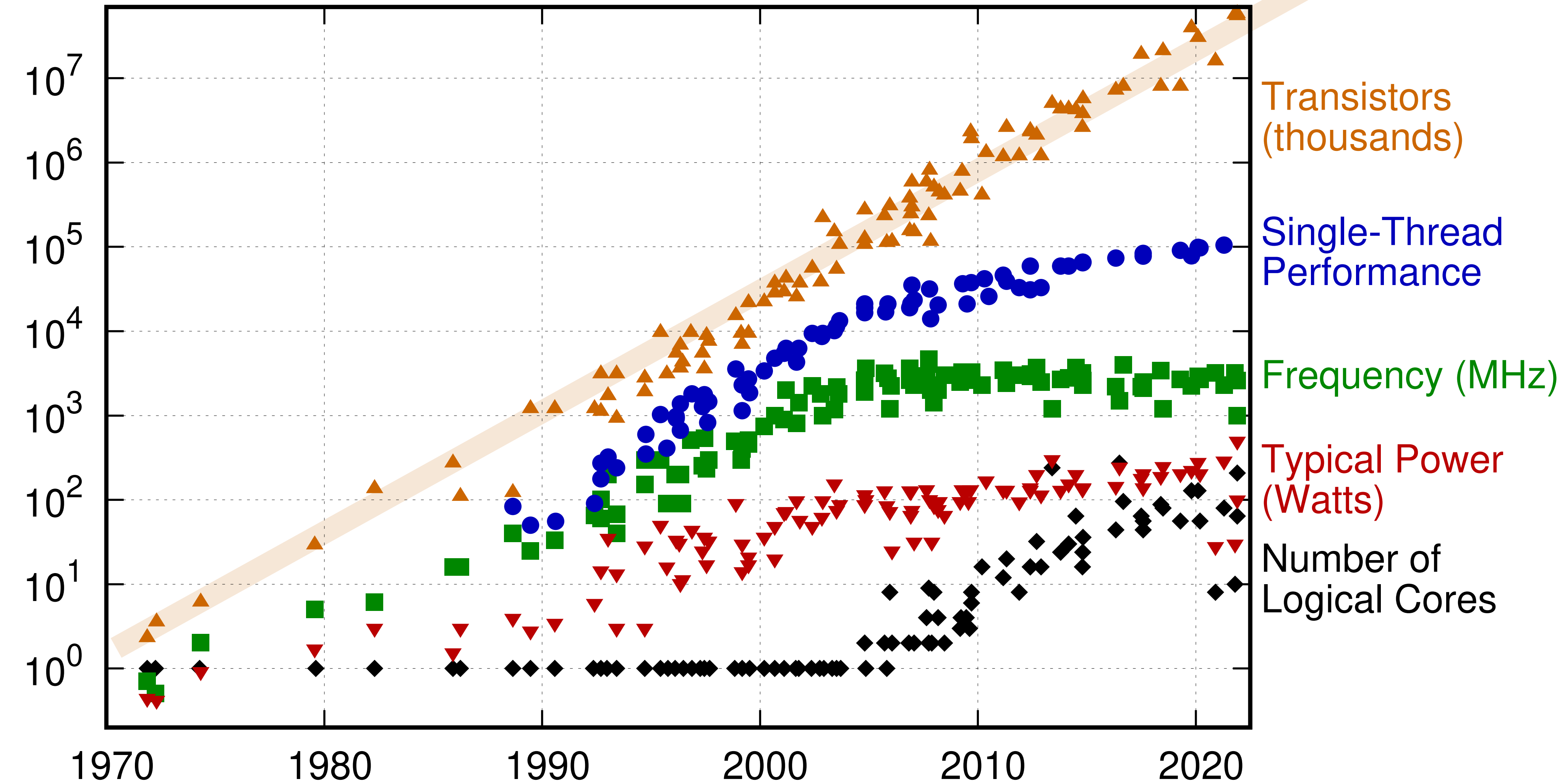


# Processor trends: 1970 → today



[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

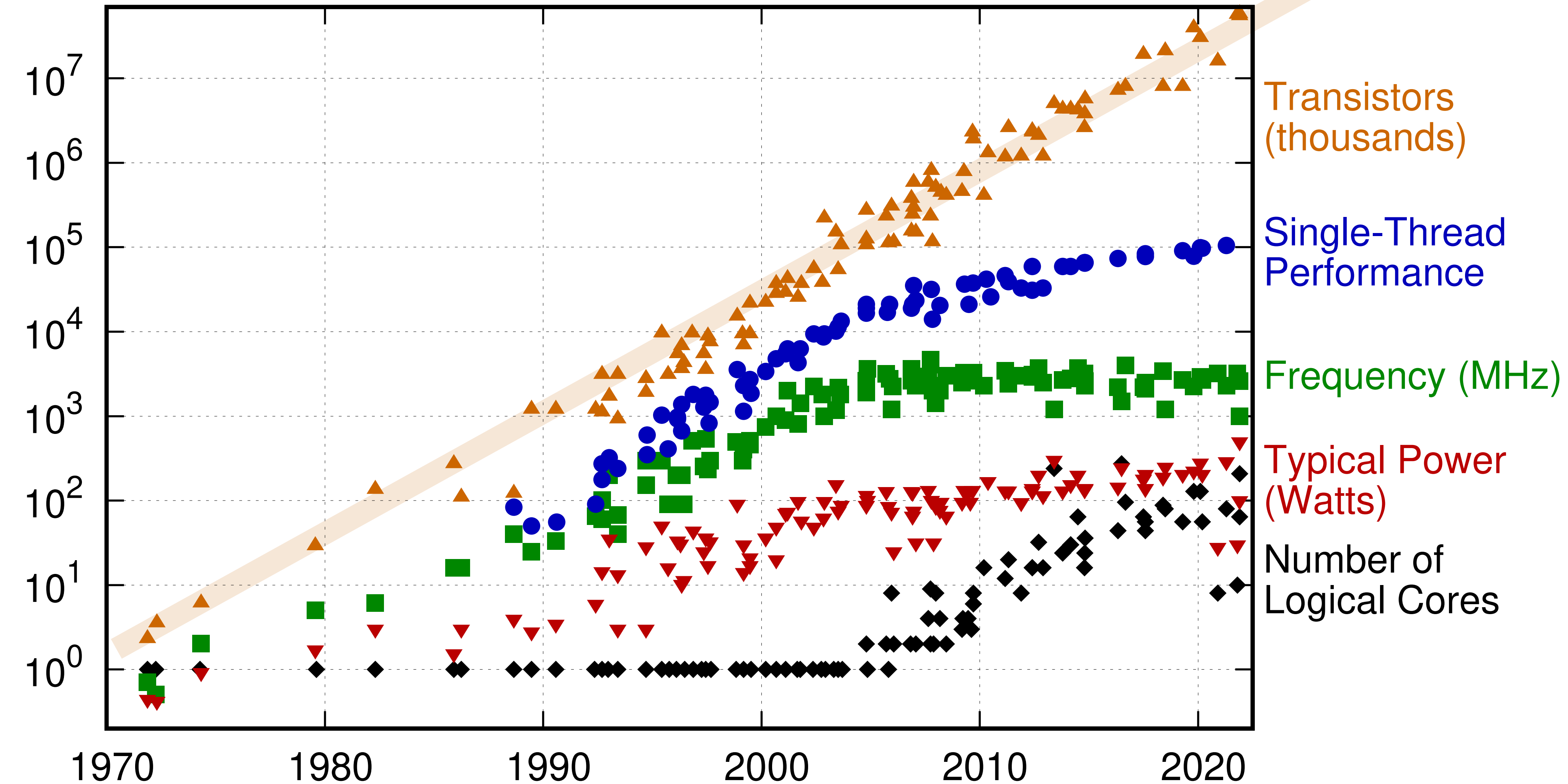
# Processor trends: 1970 → today



[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

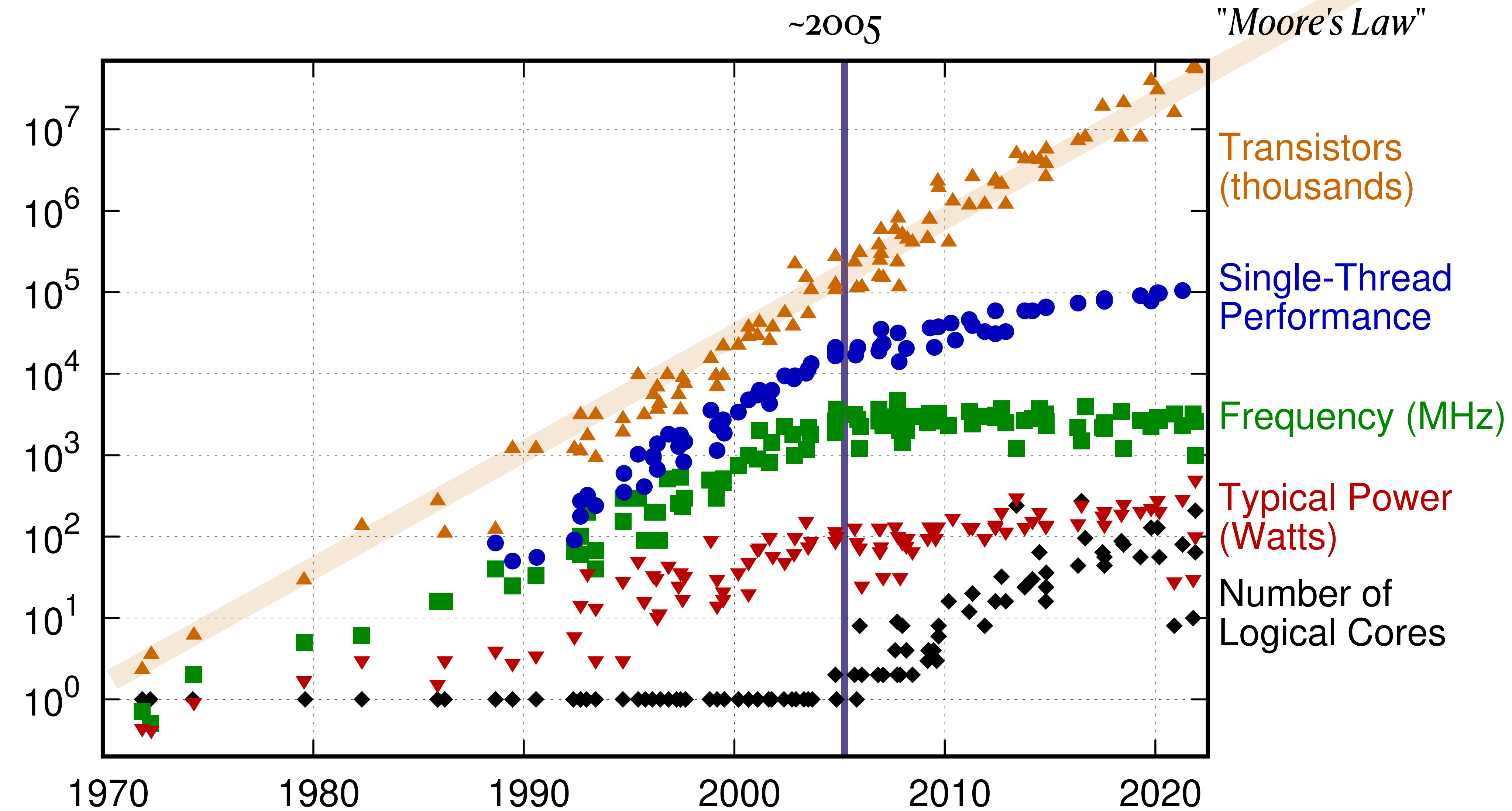
# Processor trends: 1970 → today

"Moore's Law"



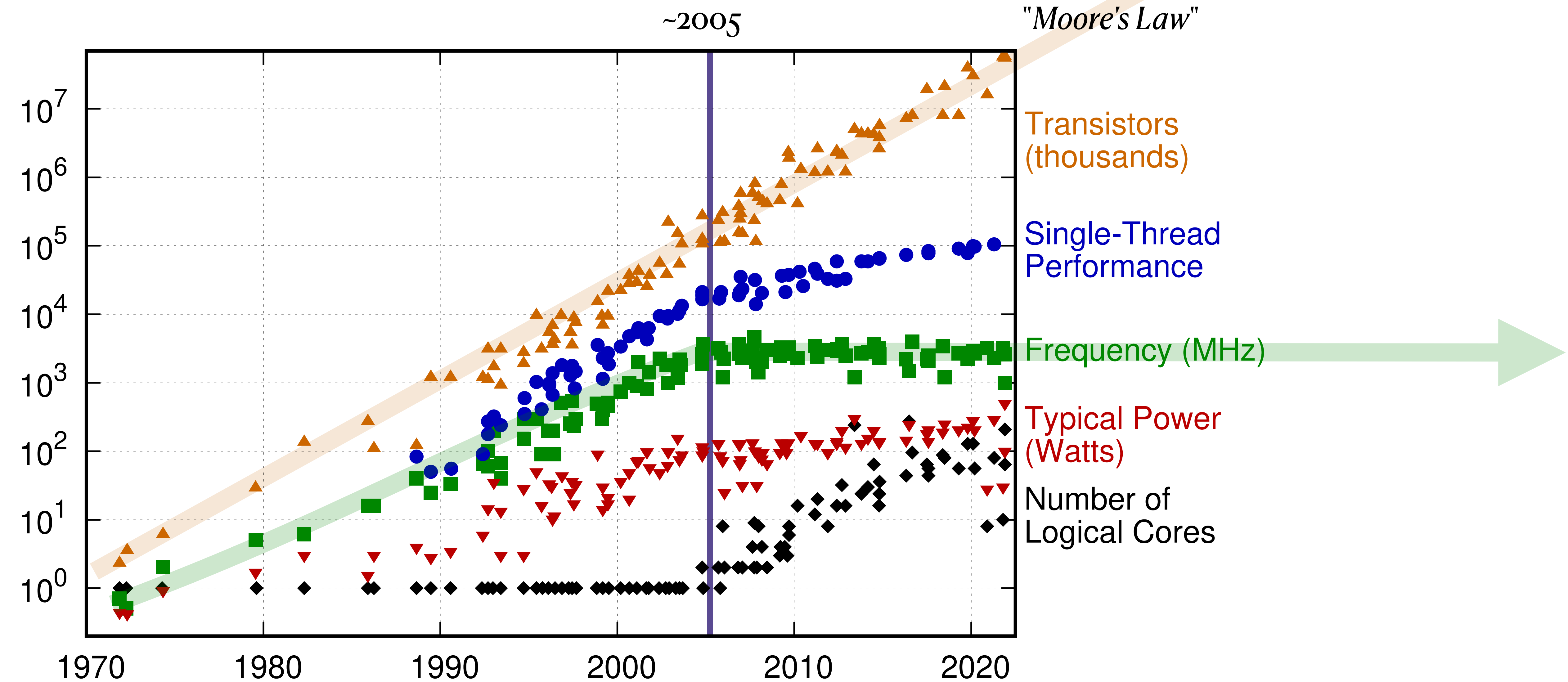
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



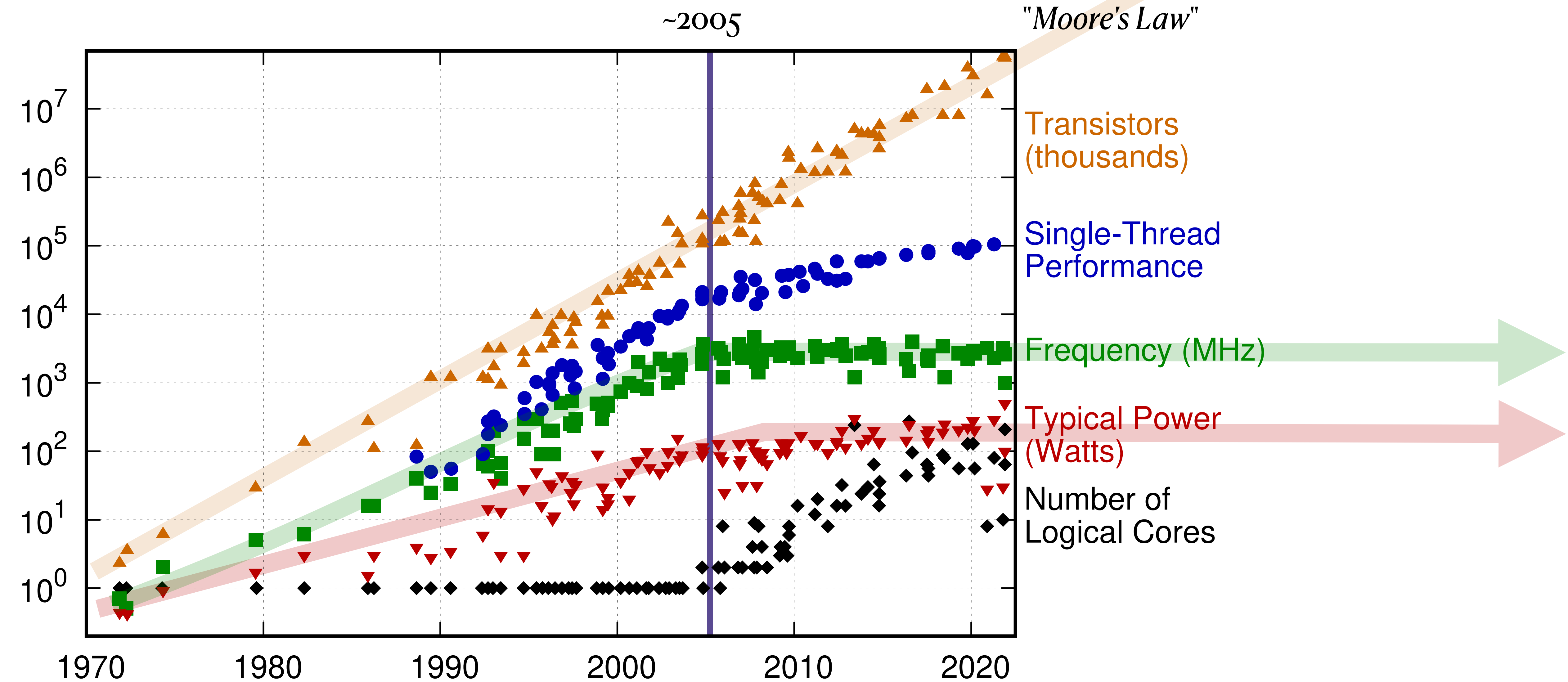
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



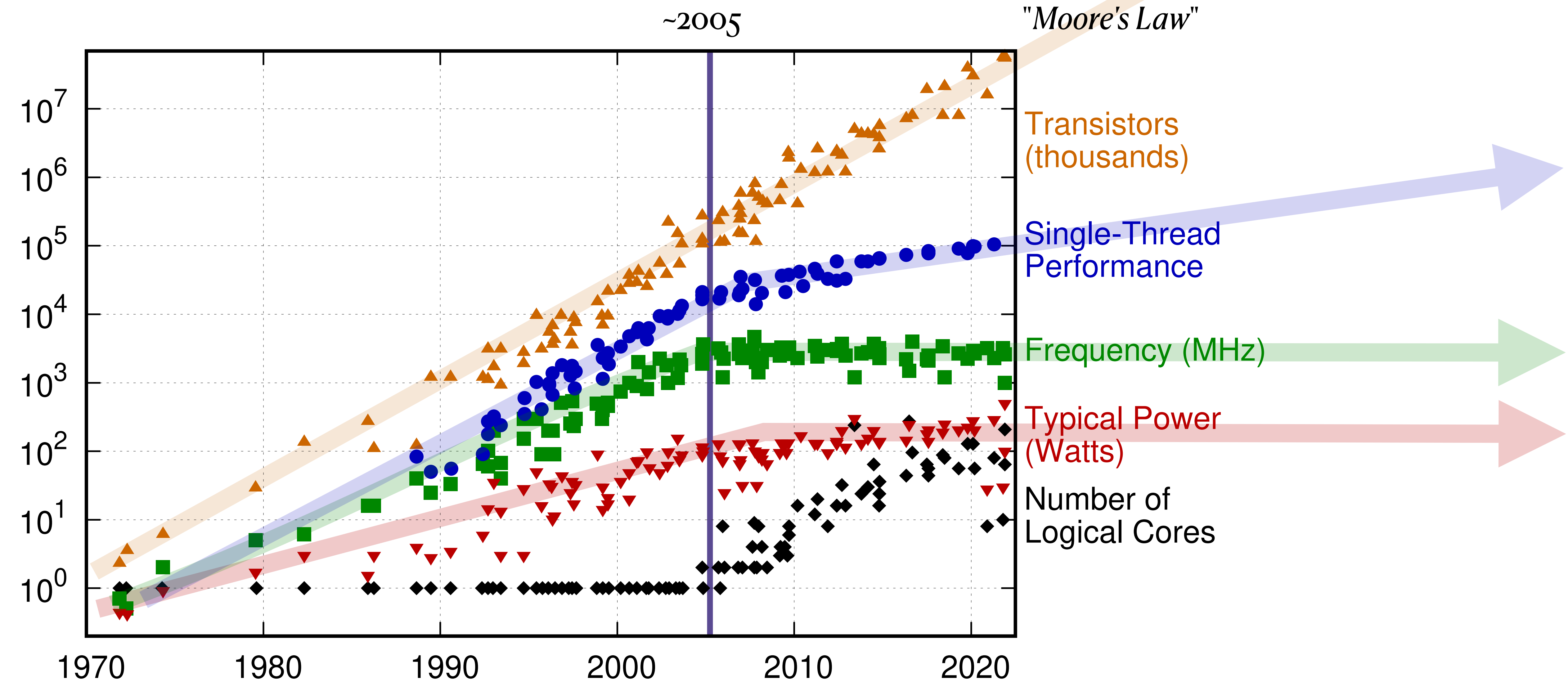
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



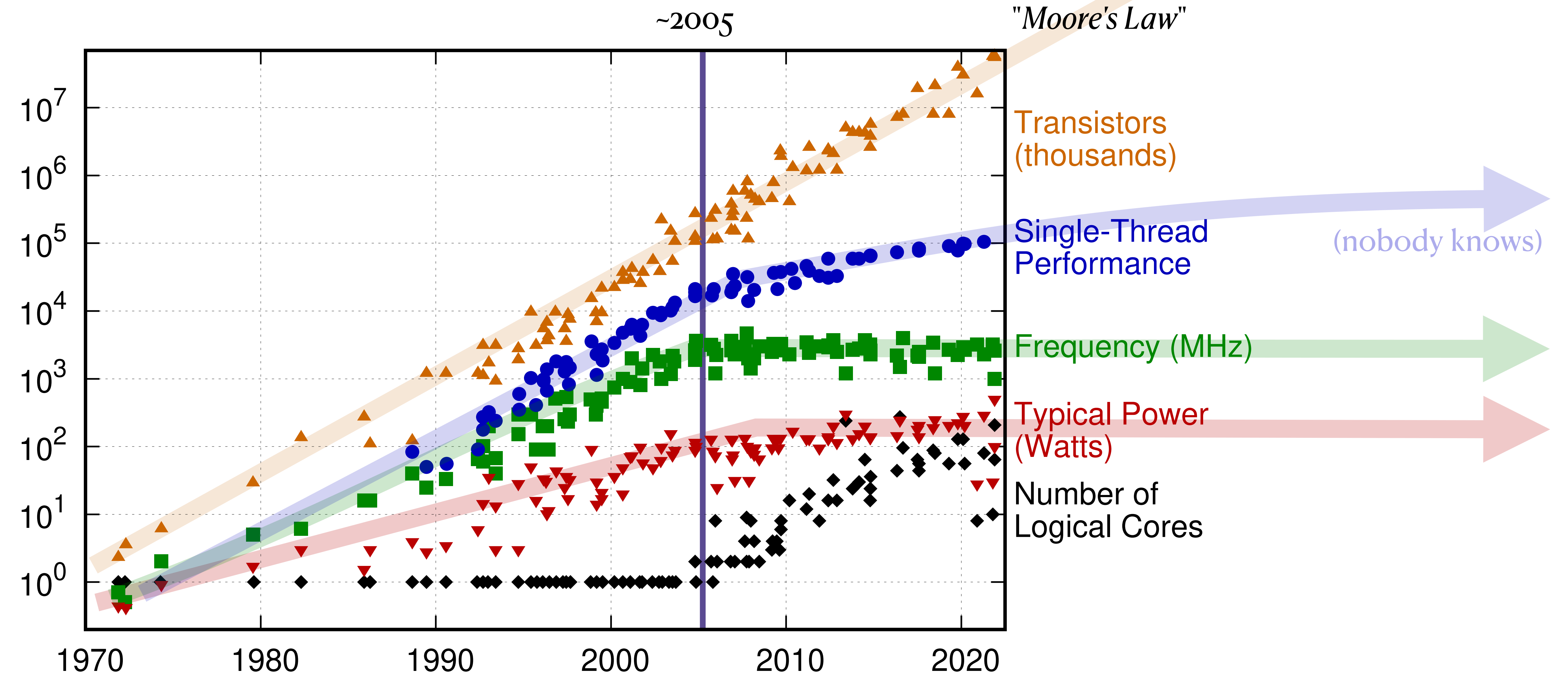
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



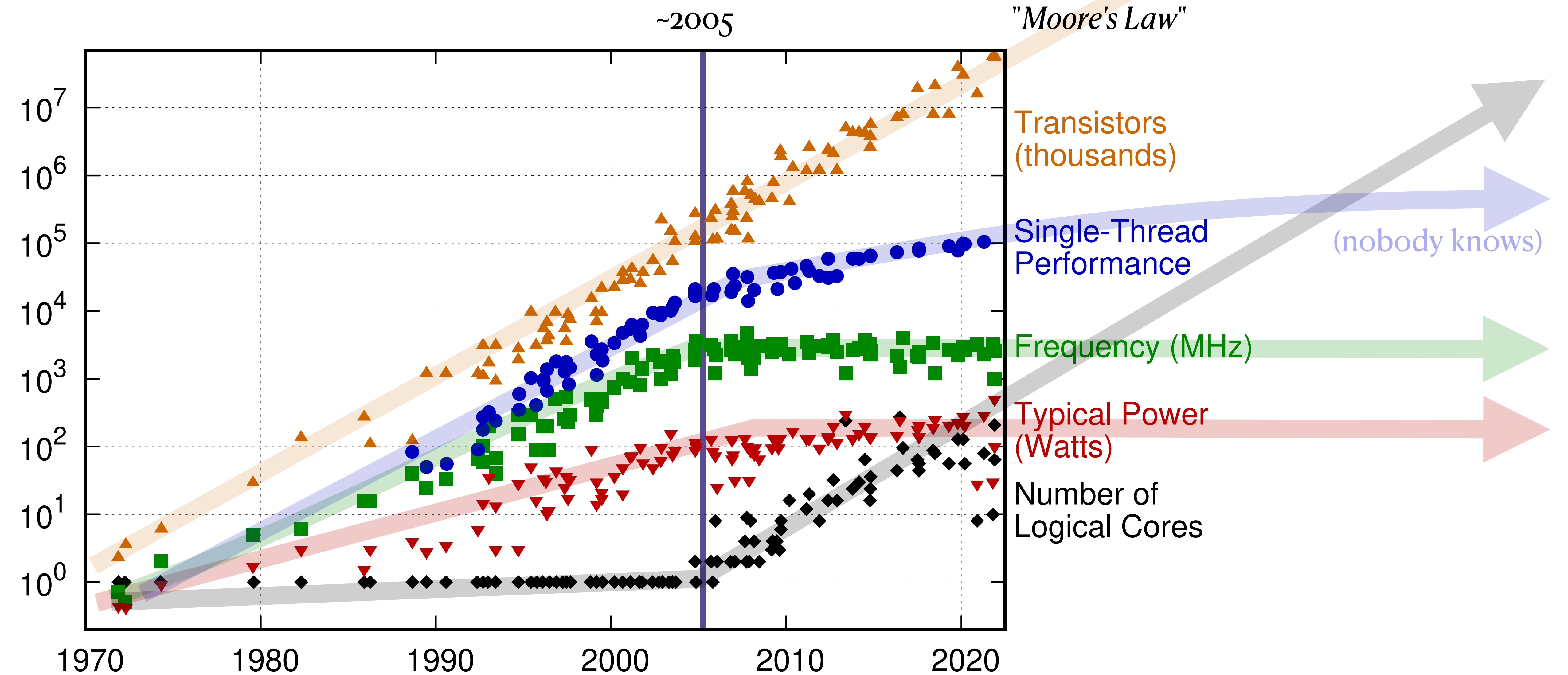
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



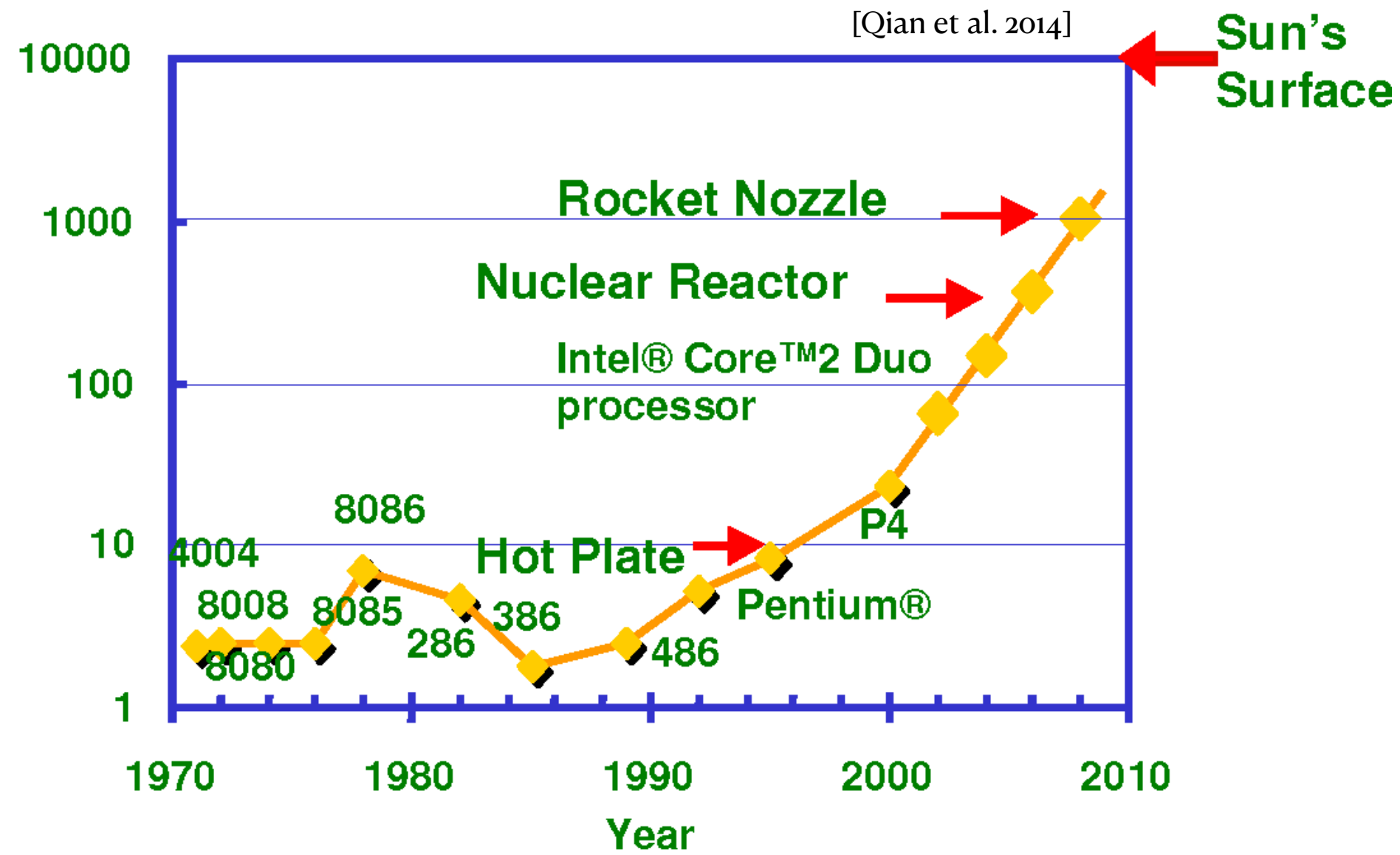
[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

# Processor trends: 1970 → today



[Source: <https://github.com/karlrupp/microprocessor-trend-data>]

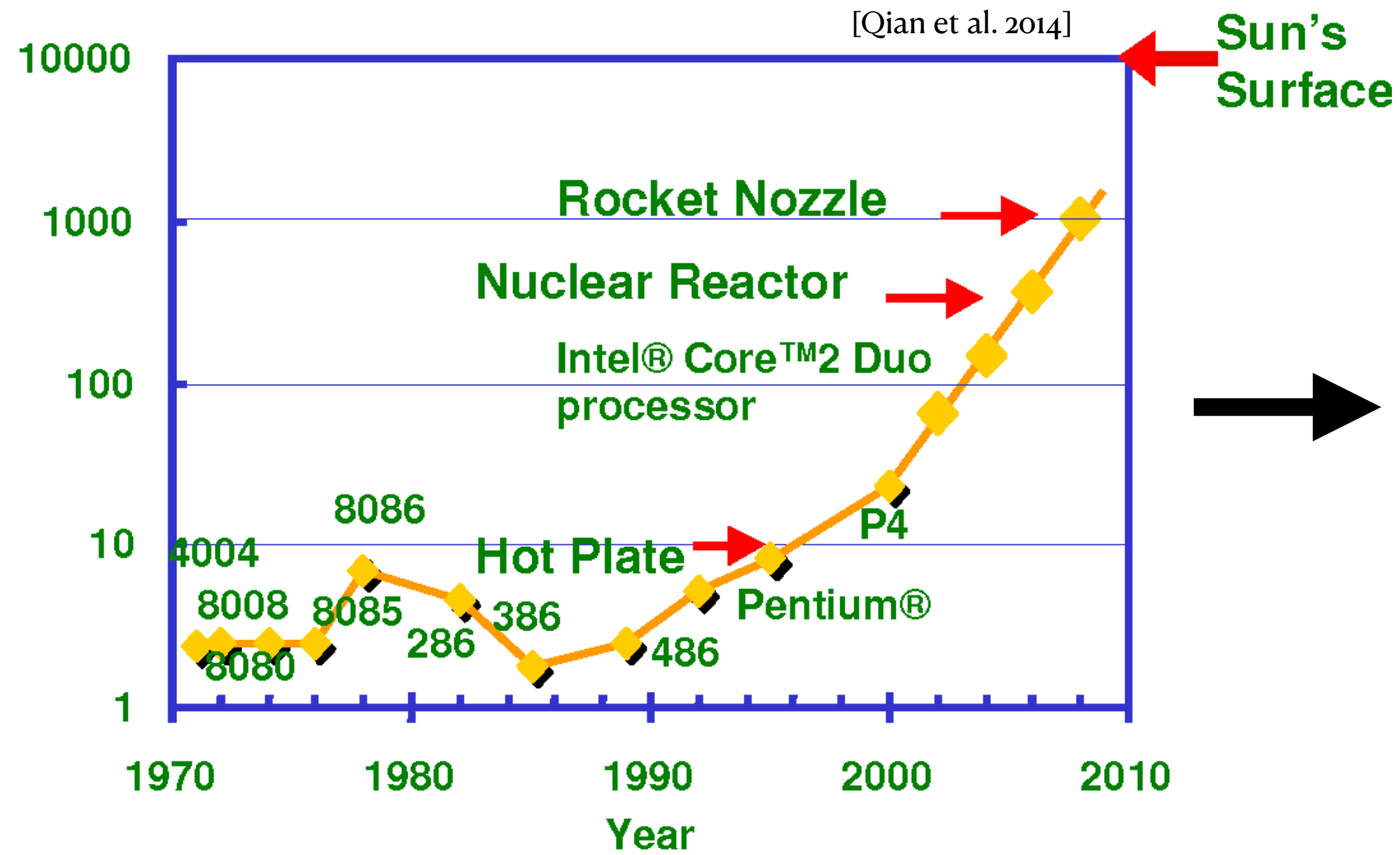
# But what does it *mean*?



We hit a power wall at  $\sim 100 \text{ W/cm}^2$

$\Rightarrow$  Frequency fixed because raising it increases power usage & temperature.

# But what does it *mean*?



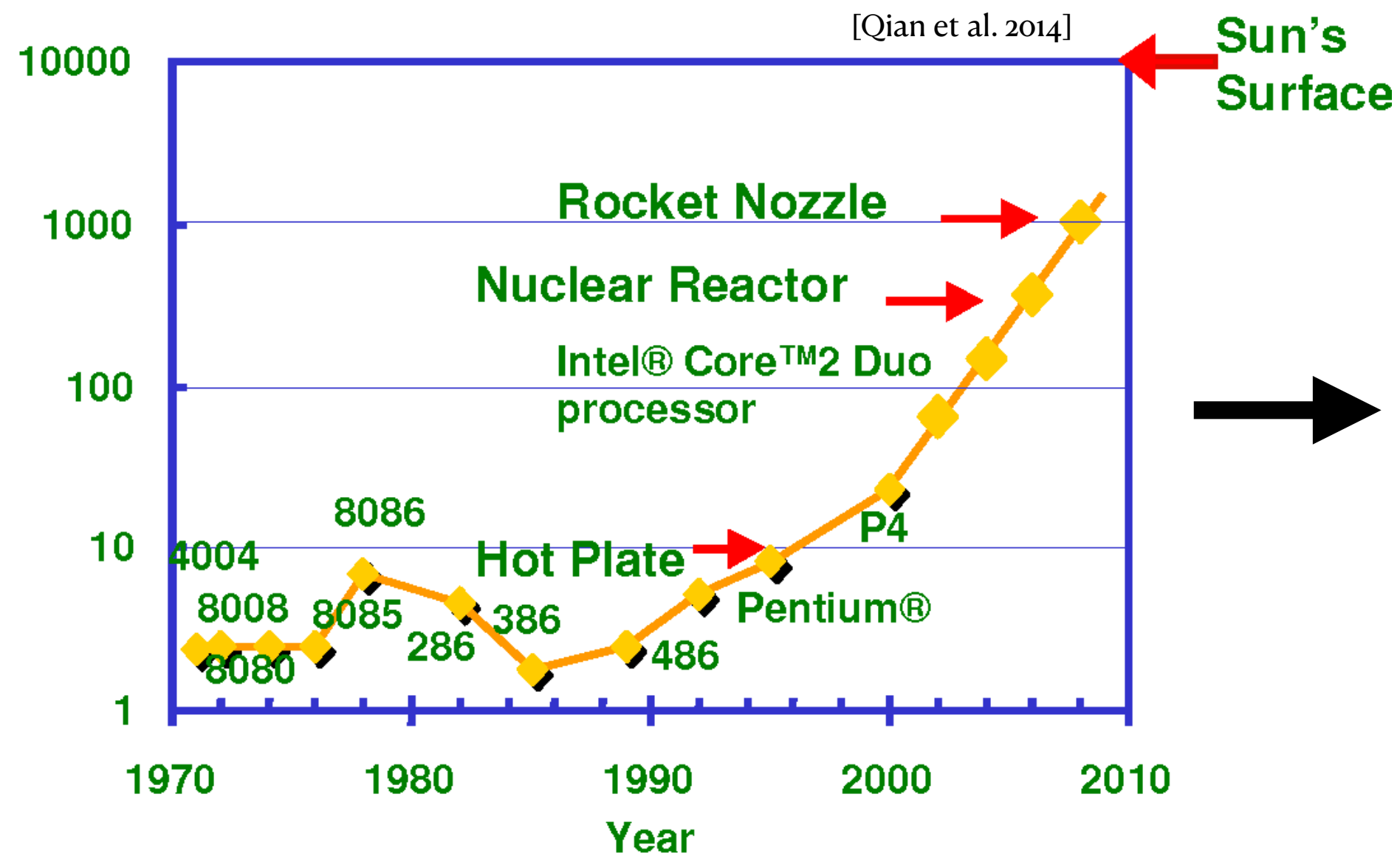
MidJourney: *a sad computer*



We hit a power wall at  $\sim 100 \text{ W/cm}^2$   
 $\Rightarrow$  Frequency fixed because raising it increases power usage & temperature.

Our processors aren't getting much faster anymore.

# But what does it *mean*?



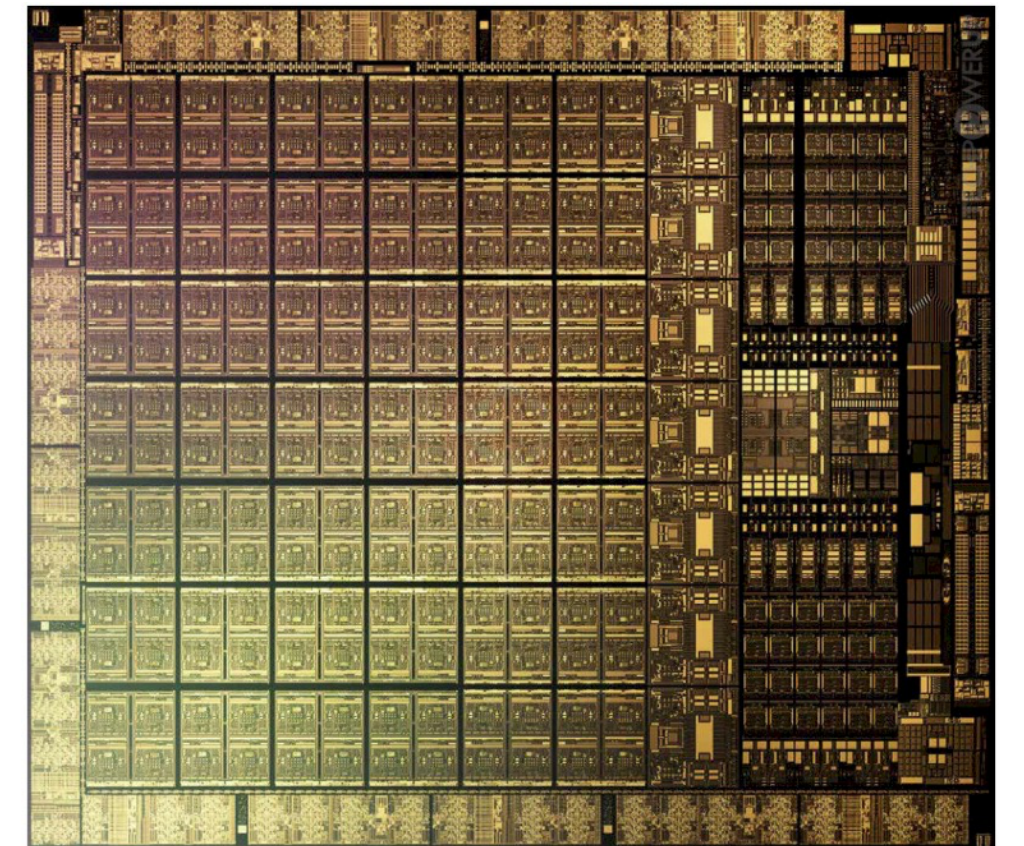
We hit a power wall at  $\sim 100 \text{ W/cm}^2$   
 $\Rightarrow$  Frequency fixed because raising it increases power usage & temperature.

MidJourney: *a sad computer*



Our processors aren't getting much faster anymore.

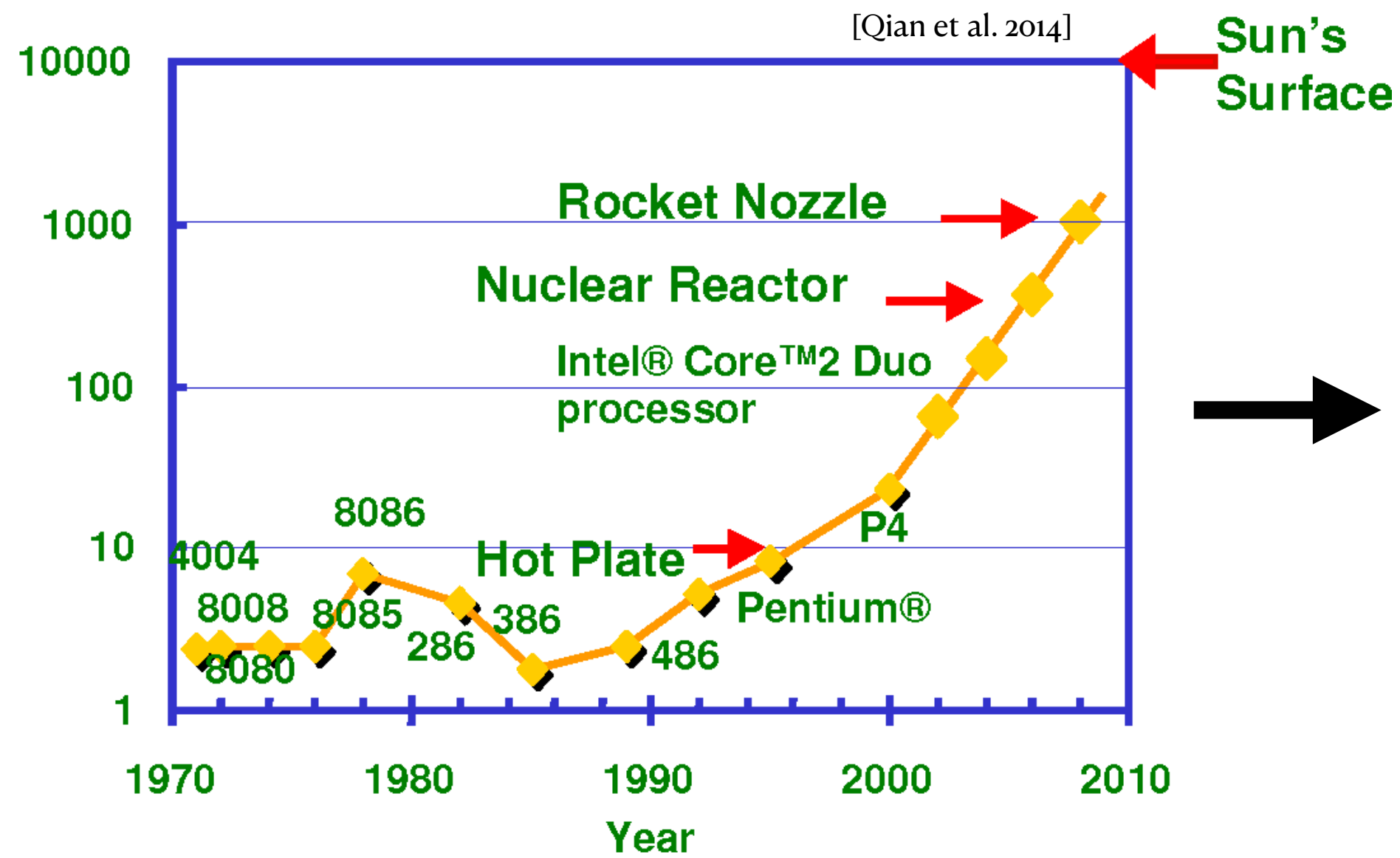
[NVIDIA]



The only thing we can do is to have *more processors*.

(Multi-core CPUs, GPUs)

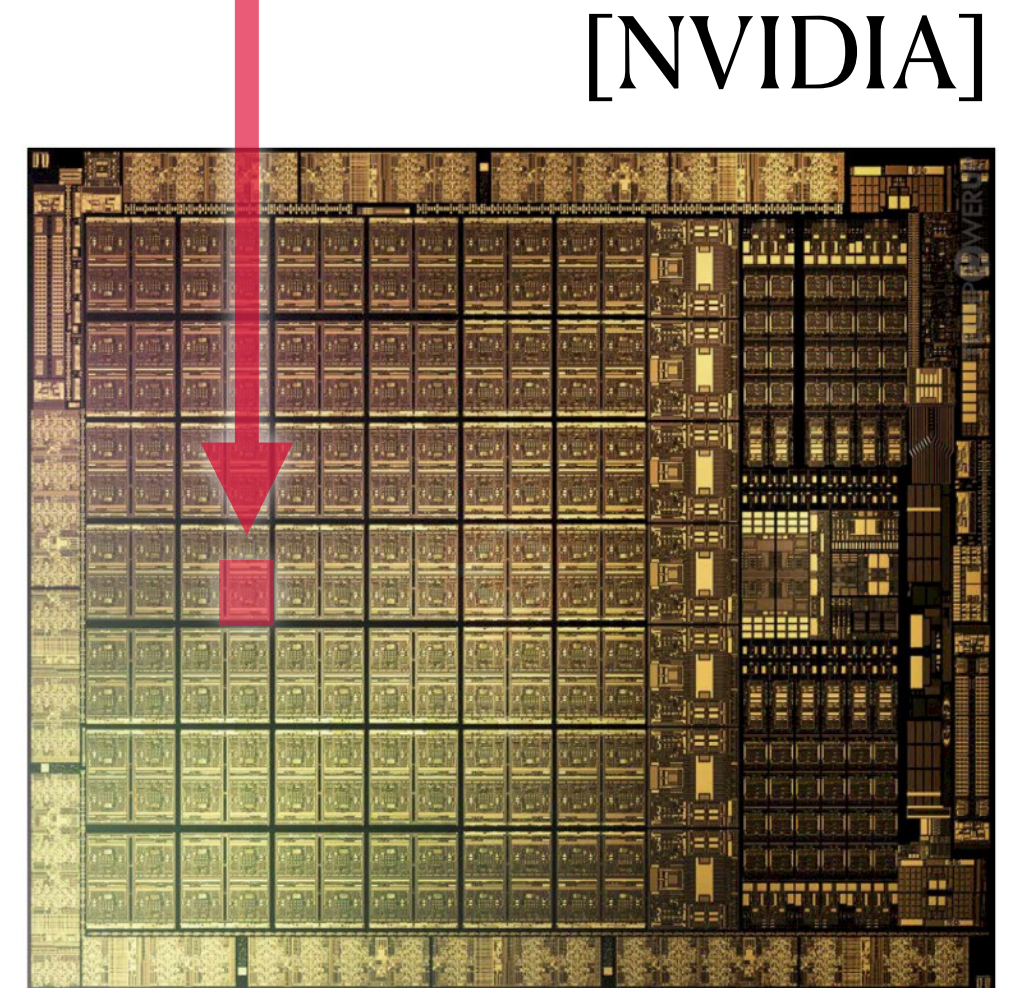
# But what does it *mean*?



MidJourney: *a sad computer*



Our processors aren't getting much faster anymore.



The only thing we can do is to have *more processors*.

(Multi-core CPUs, GPUs)

We hit a power wall at  $\sim 100 \text{ W/cm}^2$   
 $\Rightarrow$  Frequency fixed because raising it increases power usage & temperature.

# Why this history lesson?

- Only way to get good performance nowadays is to anticipate how an algorithm will run on a particular piece of hardware and then *design for it*.

# Why this history lesson?

- Only way to get good performance nowadays is to anticipate how an algorithm will run on a particular piece of hardware and then *design for it*.
- Sometimes that means using a good implementation of a common building block (e.g. matrix factorization built by Intel/NVIDIA/..). When your needs are more special, you will likely have to make it yourself.

# Why this history lesson?

- Only way to get good performance nowadays is to anticipate how an algorithm will run on a particular piece of hardware and then *design for it*.
- Sometimes that means using a good implementation of a common building block (e.g. matrix factorization built by Intel/NVIDIA/..). When your needs are more special, you will likely have to make it yourself.
- Can easily get order-of-magnitude speedups. That might mean the difference between:
  - Your project is a **smashing success**.
  - **Nobody cares** about what you do.

# Why this history lesson?

- Only way to get good performance nowadays is to anticipate how an algorithm will run on a particular piece of hardware and then *design for it*.
- Sometimes that means using a good implementation of a common building block (e.g. matrix factorization built by Intel/NVIDIA/..). When your needs are more special, you will likely have to make it yourself.
- Can easily get order-of-magnitude speedups. That might mean the difference between:
  - Your project is a **smashing success**.
  - **Nobody cares** about what you do.
- Important even if you don't care about "low level stuff". (@SysCom students 😊)

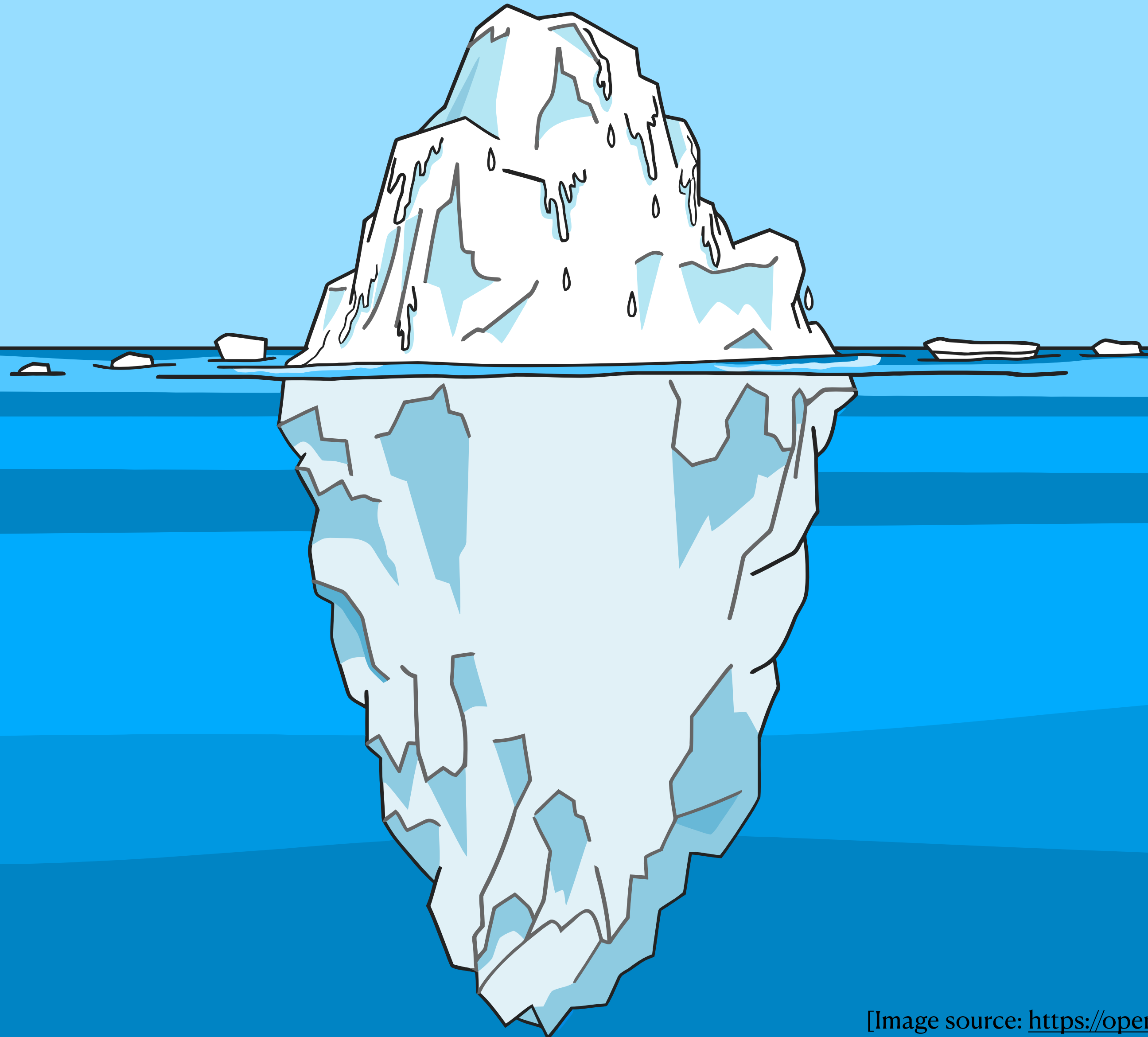
# Today's lecture

- This lecture is a **deep dive** into the performance of a simple operation.
- If you haven't seen such things before and it is all a little overwhelming:
  - *Don't be scared.*
    - You don't have to learn C++ or assembly code for the CS328 final exam.
  - *Don't look away.*
    - Still, you should know that these things exist, and how they *~roughly~* work.



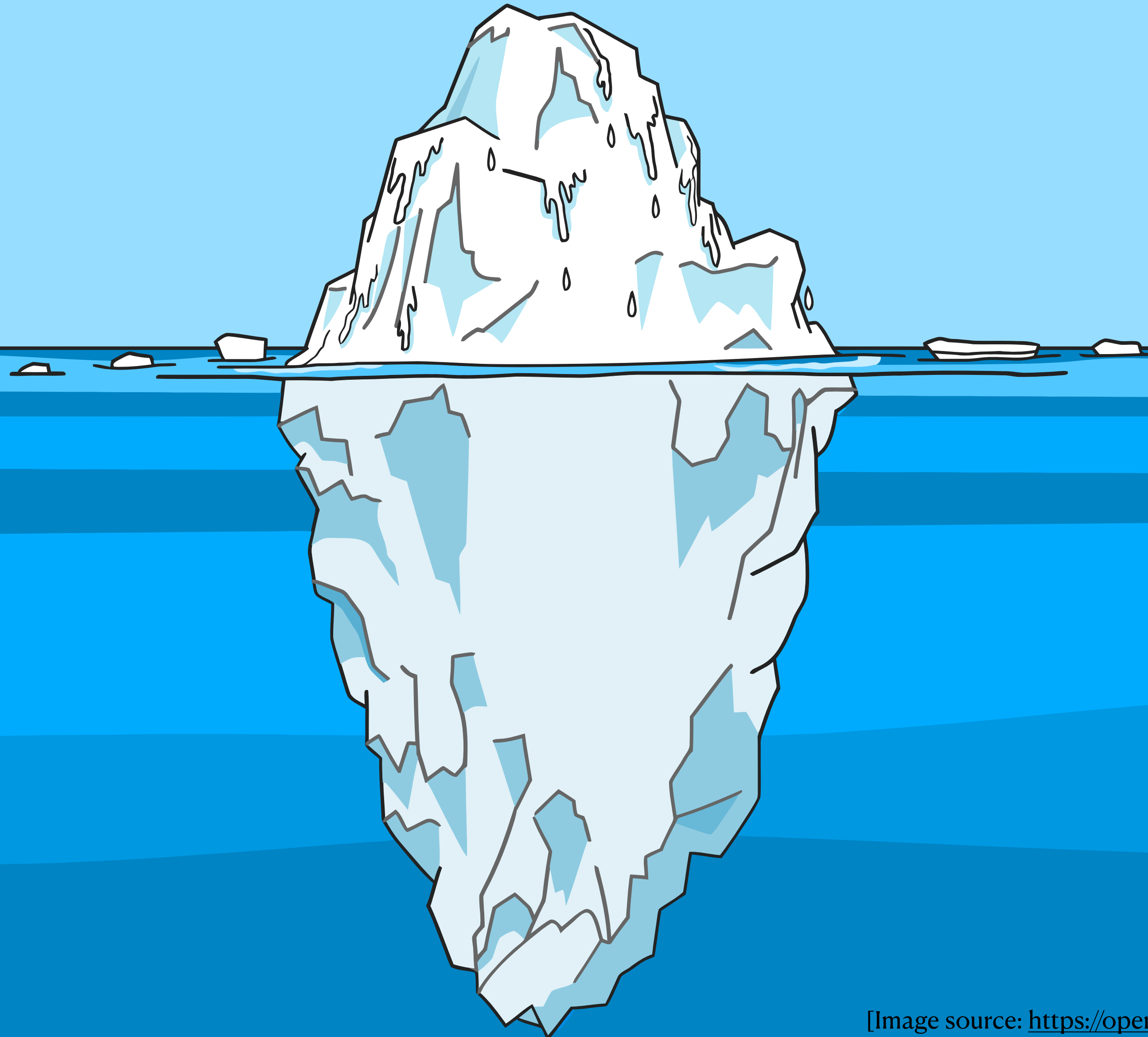
MidJourney: a deep sea diver in the ocean

# Tip of the iceberg



# Tip of the iceberg

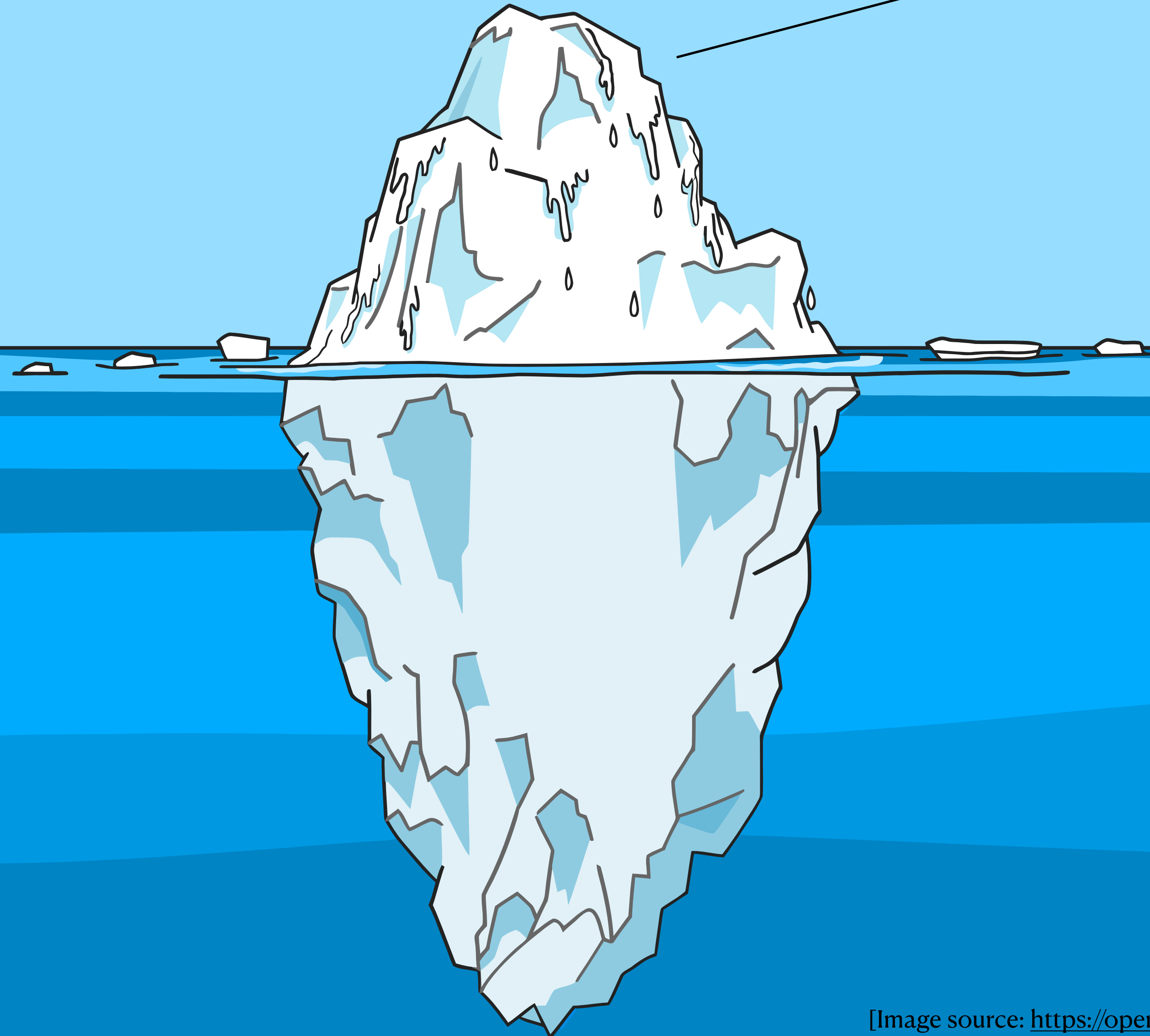
Browser



# Tip of the iceberg

Browser

Jupyter notebook process

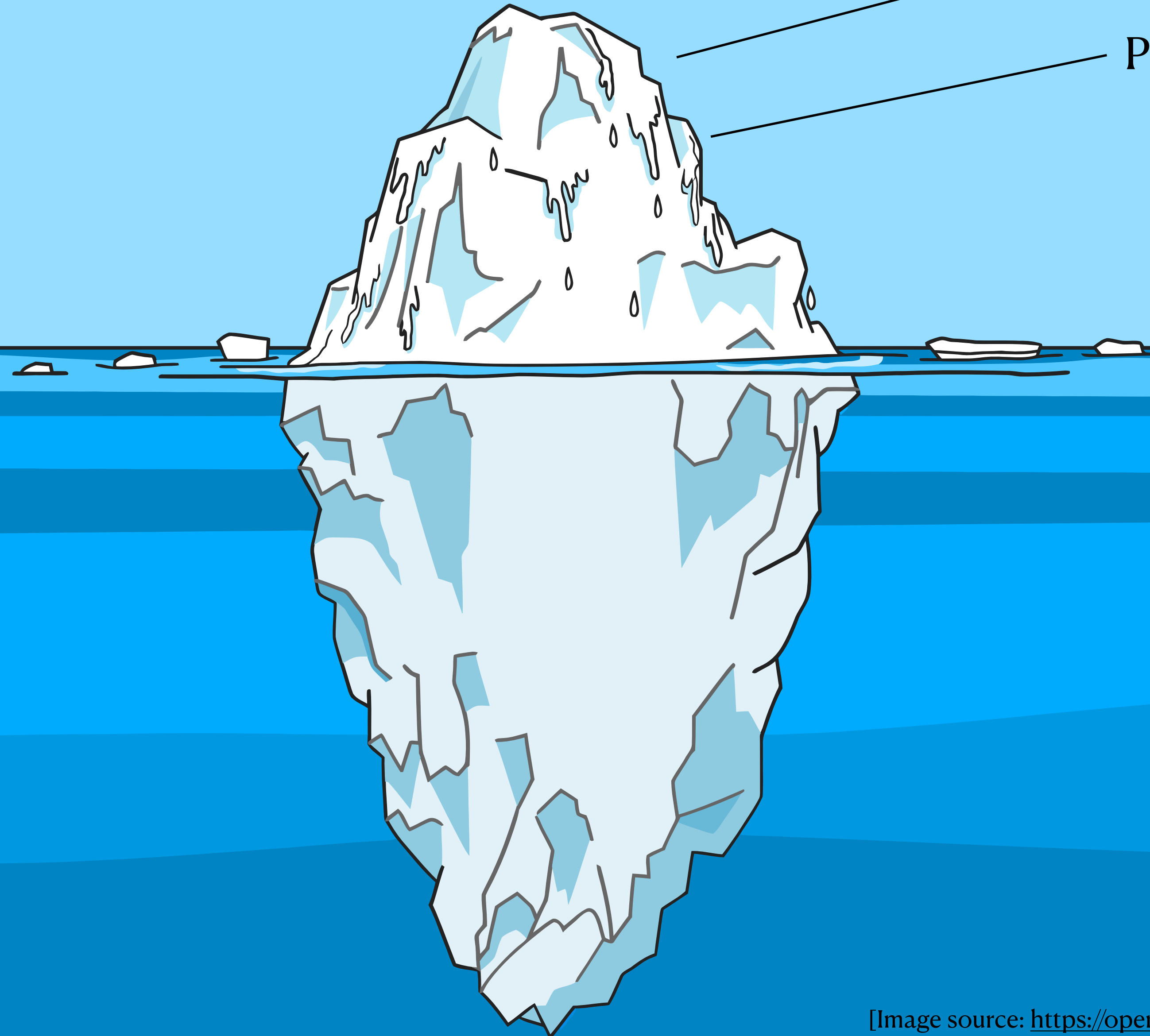


# Tip of the iceberg

Browser

Jupyter notebook process

Python framework: NumPy, ..



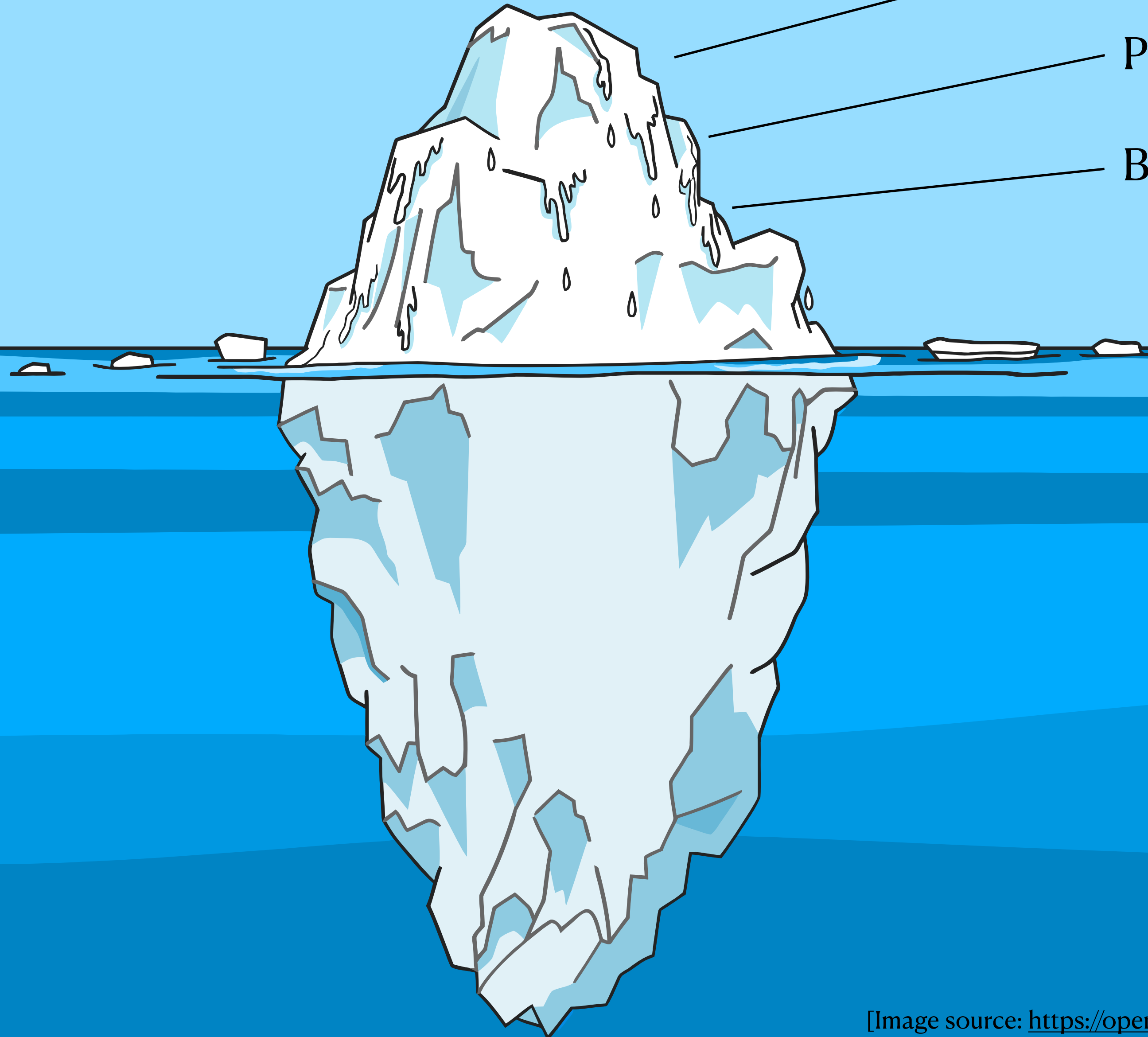
# Tip of the iceberg

Browser

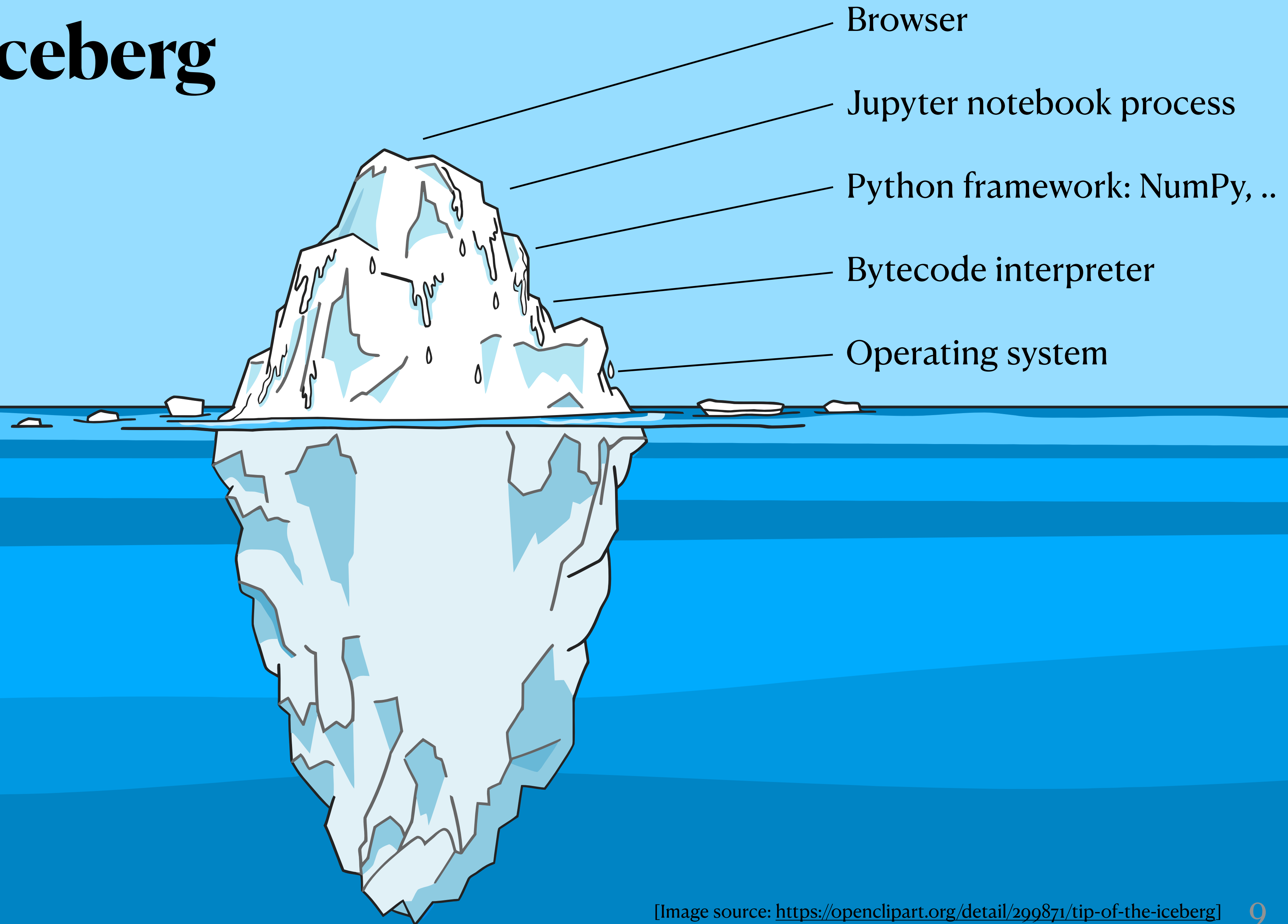
Jupyter notebook process

Python framework: NumPy, ..

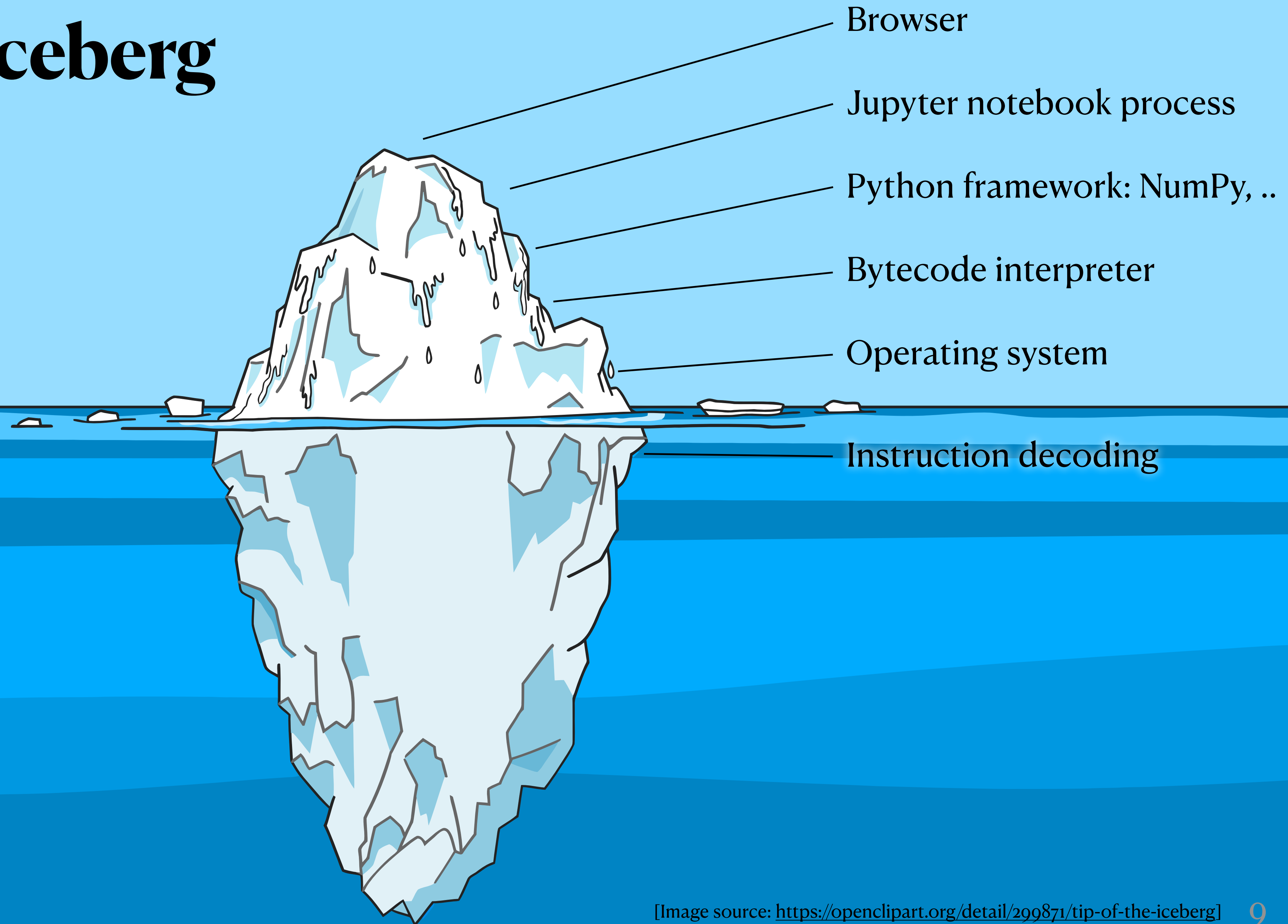
Bytecode interpreter



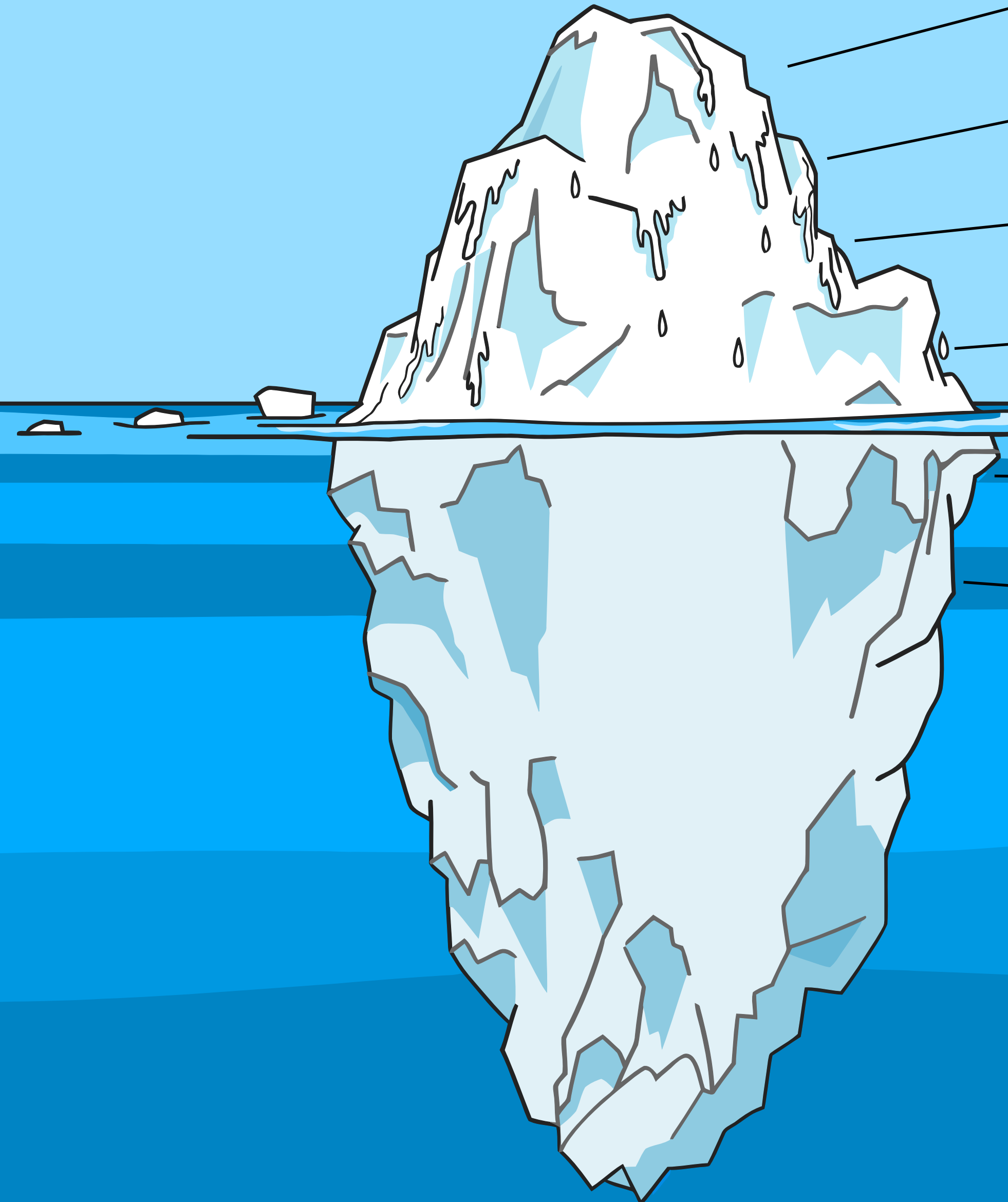
# Tip of the iceberg



# Tip of the iceberg



# Tip of the iceberg



Browser

Jupyter notebook process

Python framework: NumPy, ..

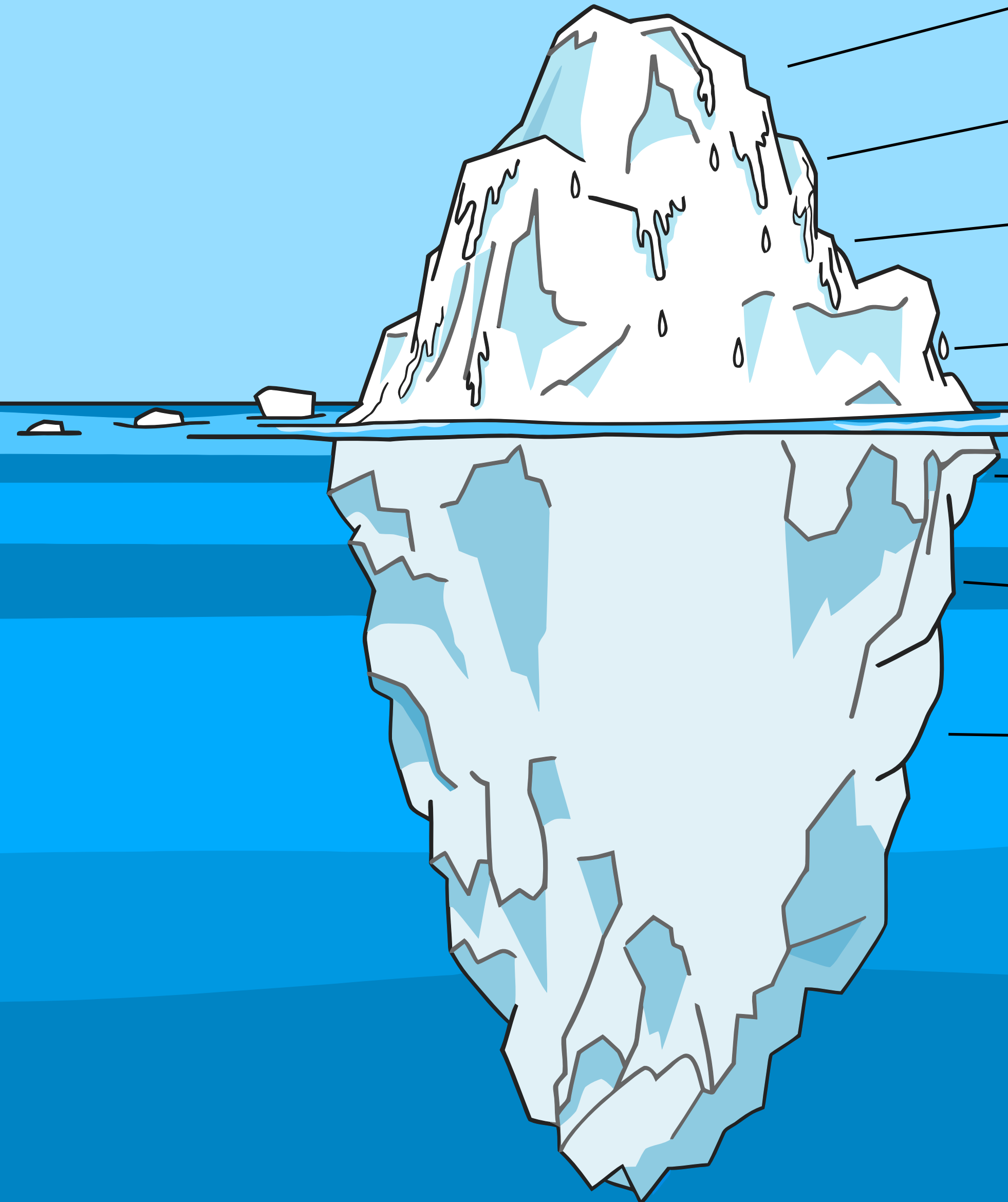
Bytecode interpreter

Operating system

Instruction decoding

Tomasulo's algorithm

# Tip of the iceberg



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

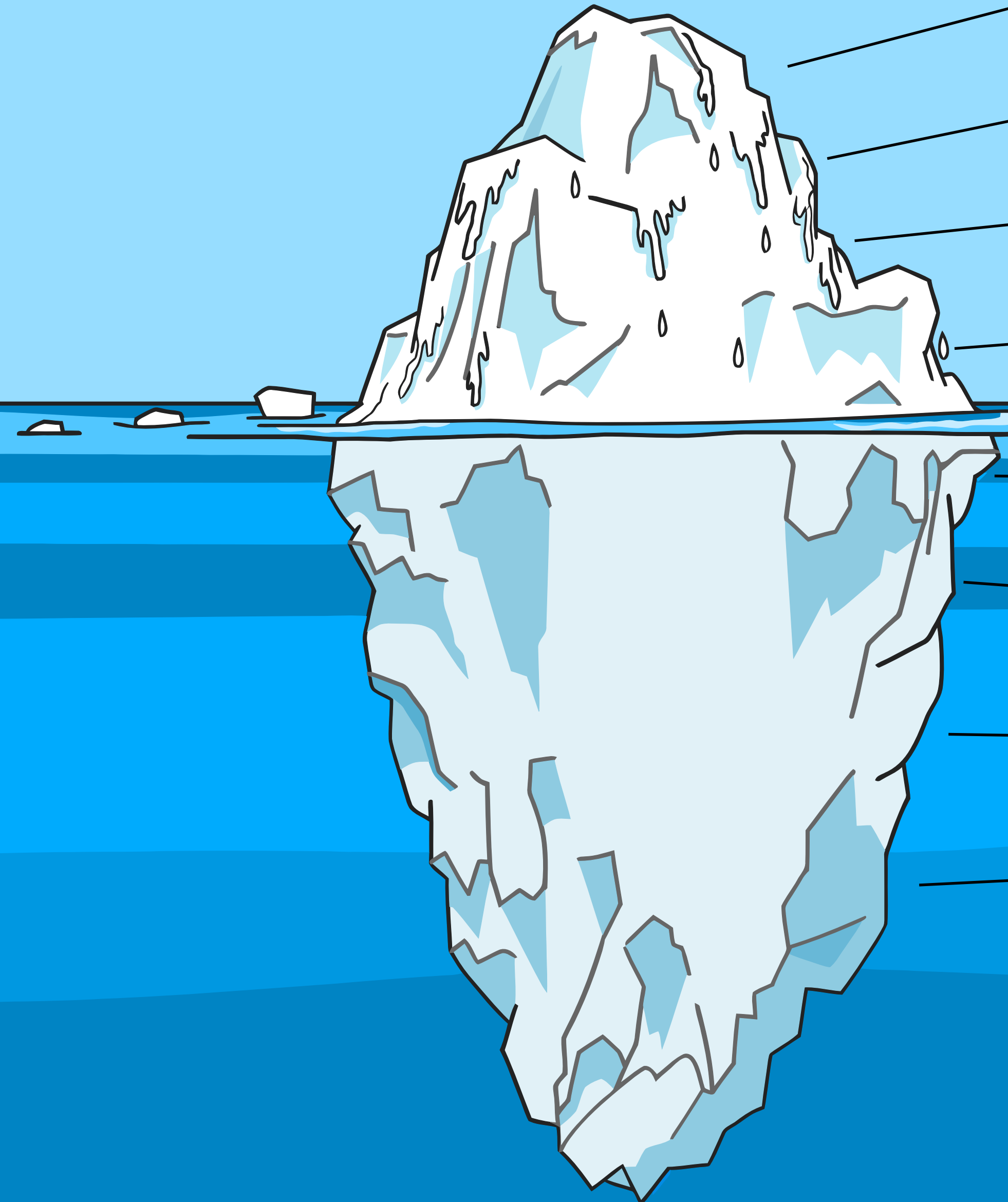
Operating system

Instruction decoding

Tomasulo's algorithm

Speculative execution

# Tip of the iceberg



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

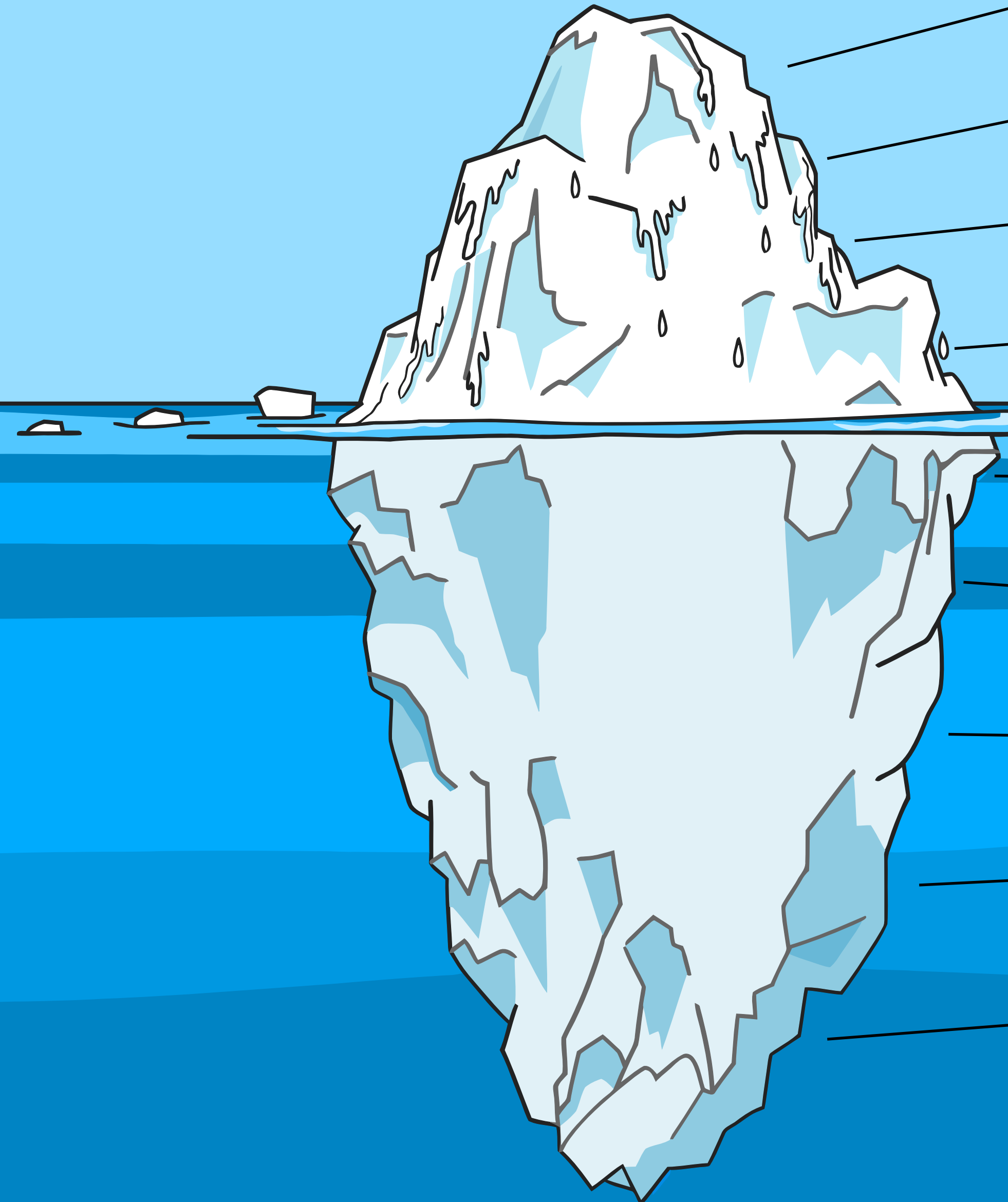
Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

# Tip of the iceberg



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

Instruction decoding

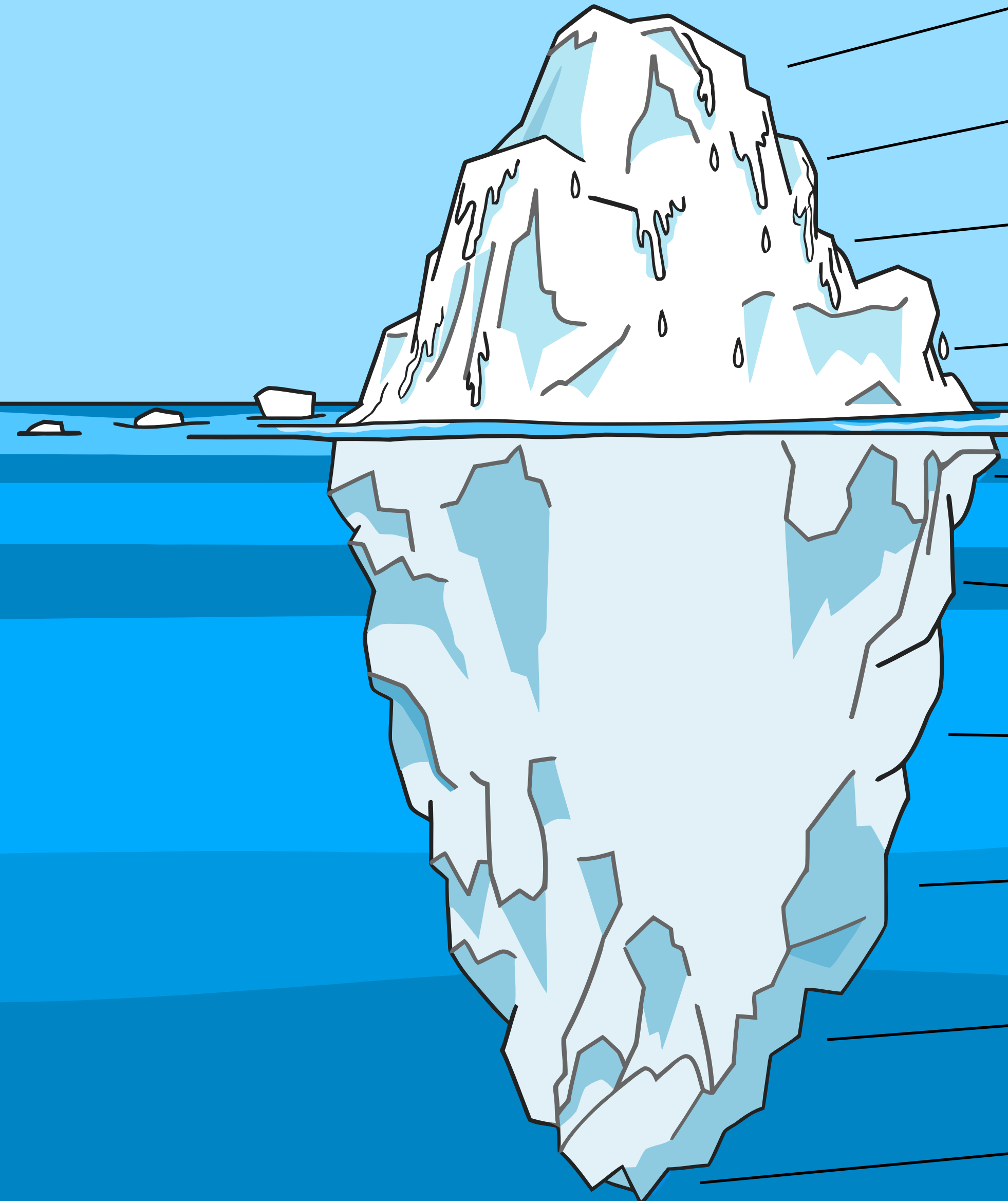
Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

DRAM

# Tip of the iceberg



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

DRAM

Physical bits

# Tip of the iceberg

# CPU

# GPU

Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

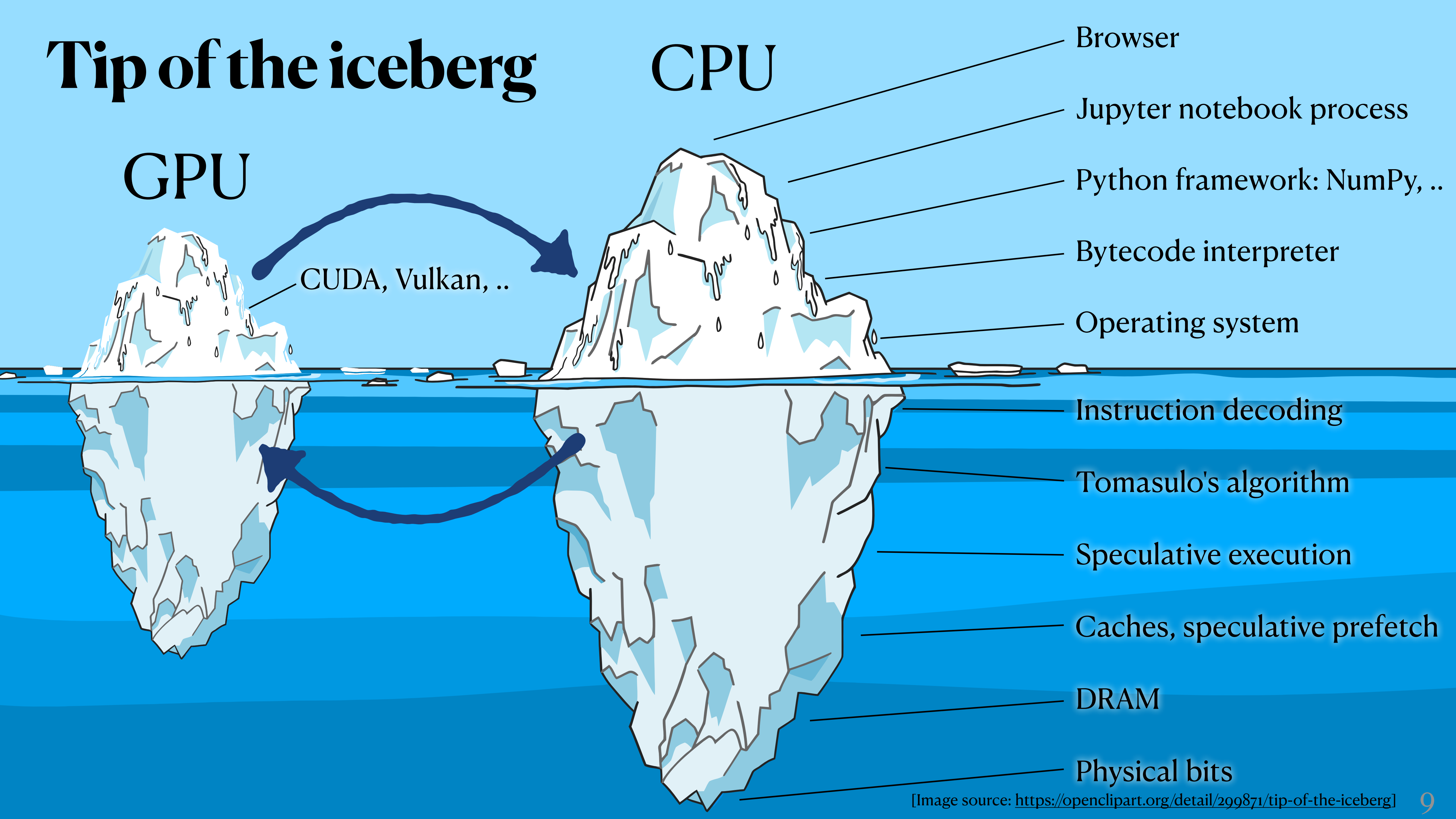
DRAM

Physical bits

# Tip of the iceberg

## CPU

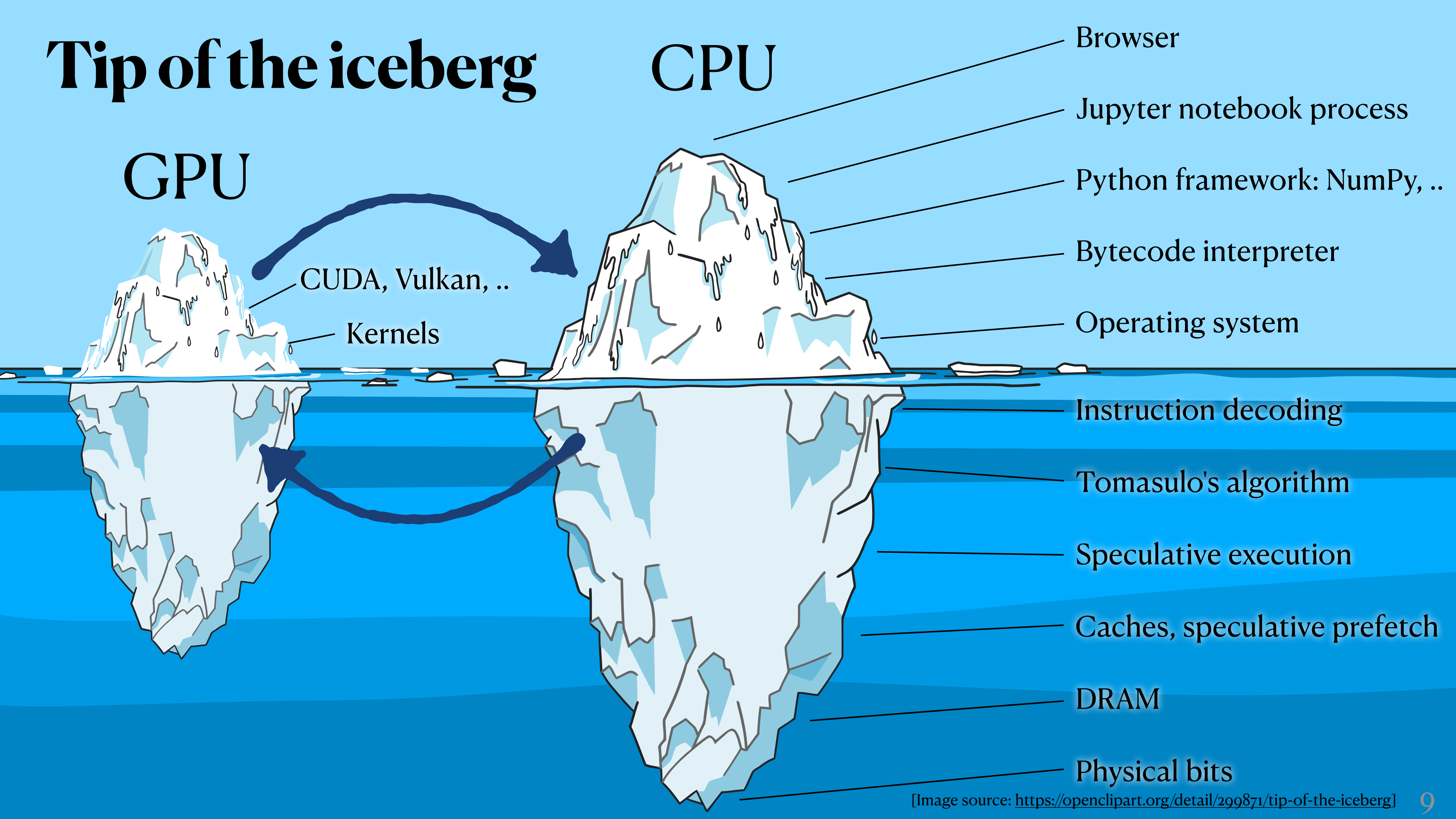
## GPU



# Tip of the iceberg

## CPU

## GPU



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

DRAM

Physical bits

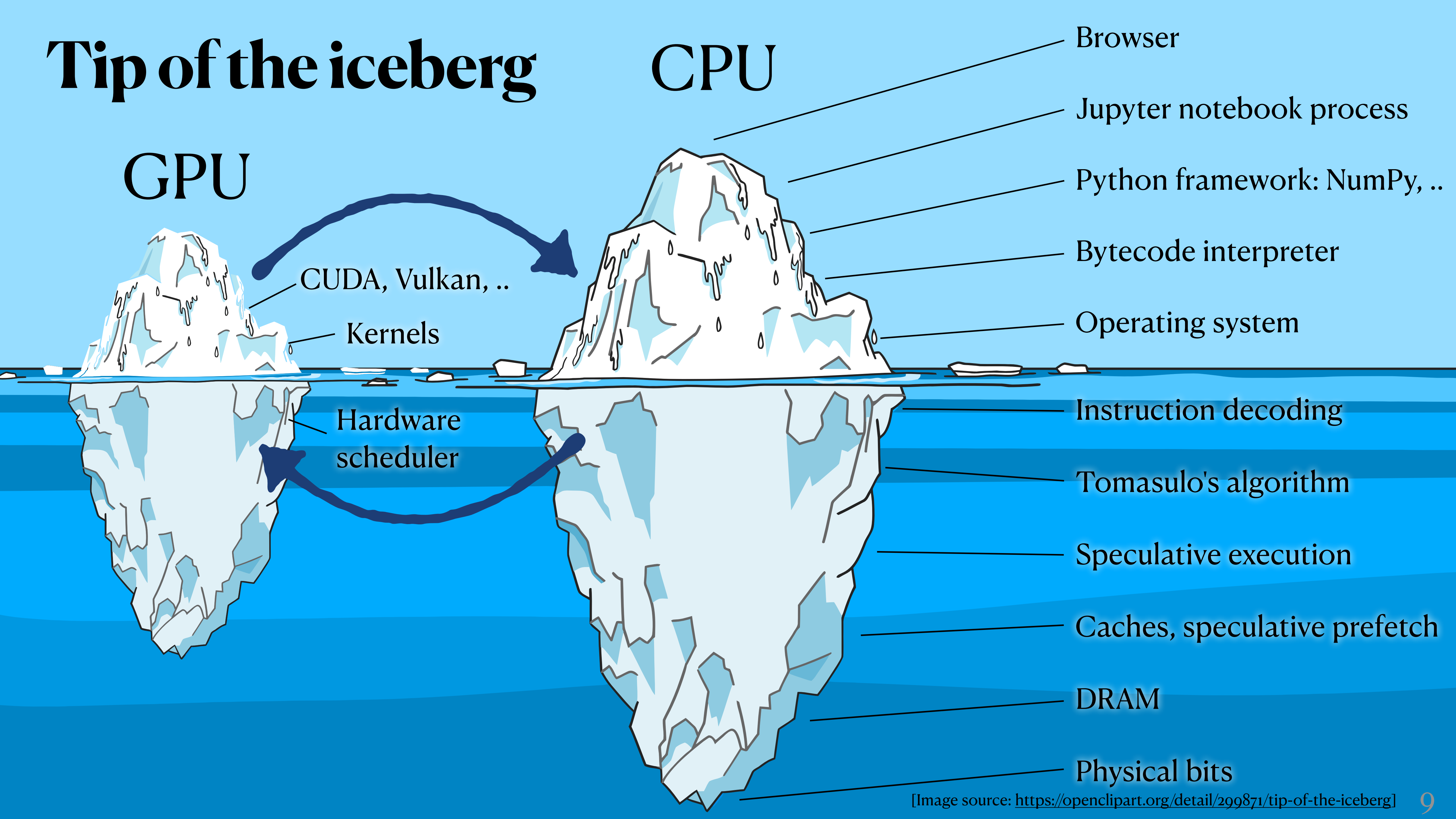
CUDA, Vulkan, ..

Kernels

# Tip of the iceberg

## CPU

## GPU



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

CUDA, Vulkan, ..

Kernels

Hardware scheduler

Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

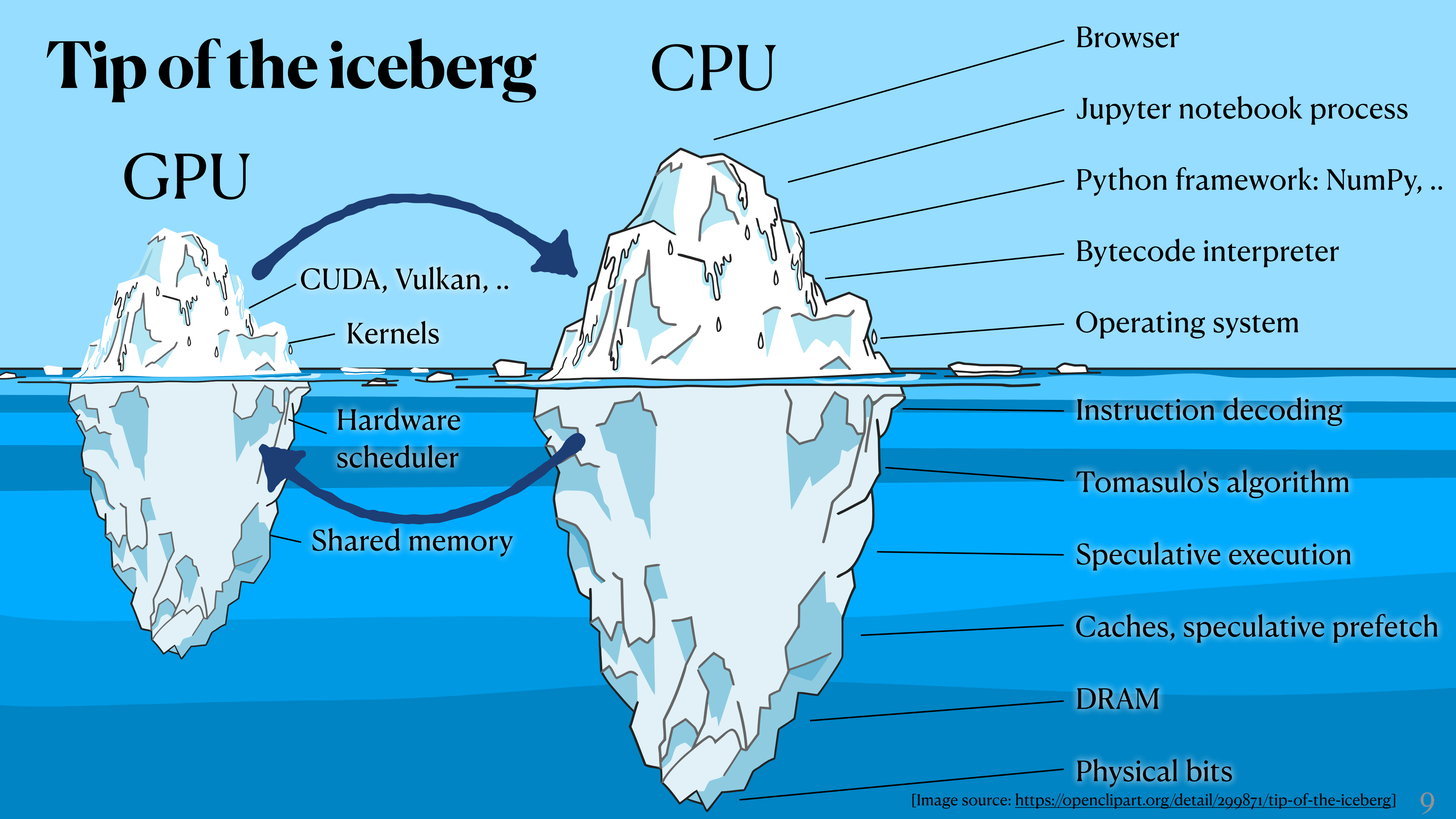
DRAM

Physical bits

# Tip of the iceberg

## CPU

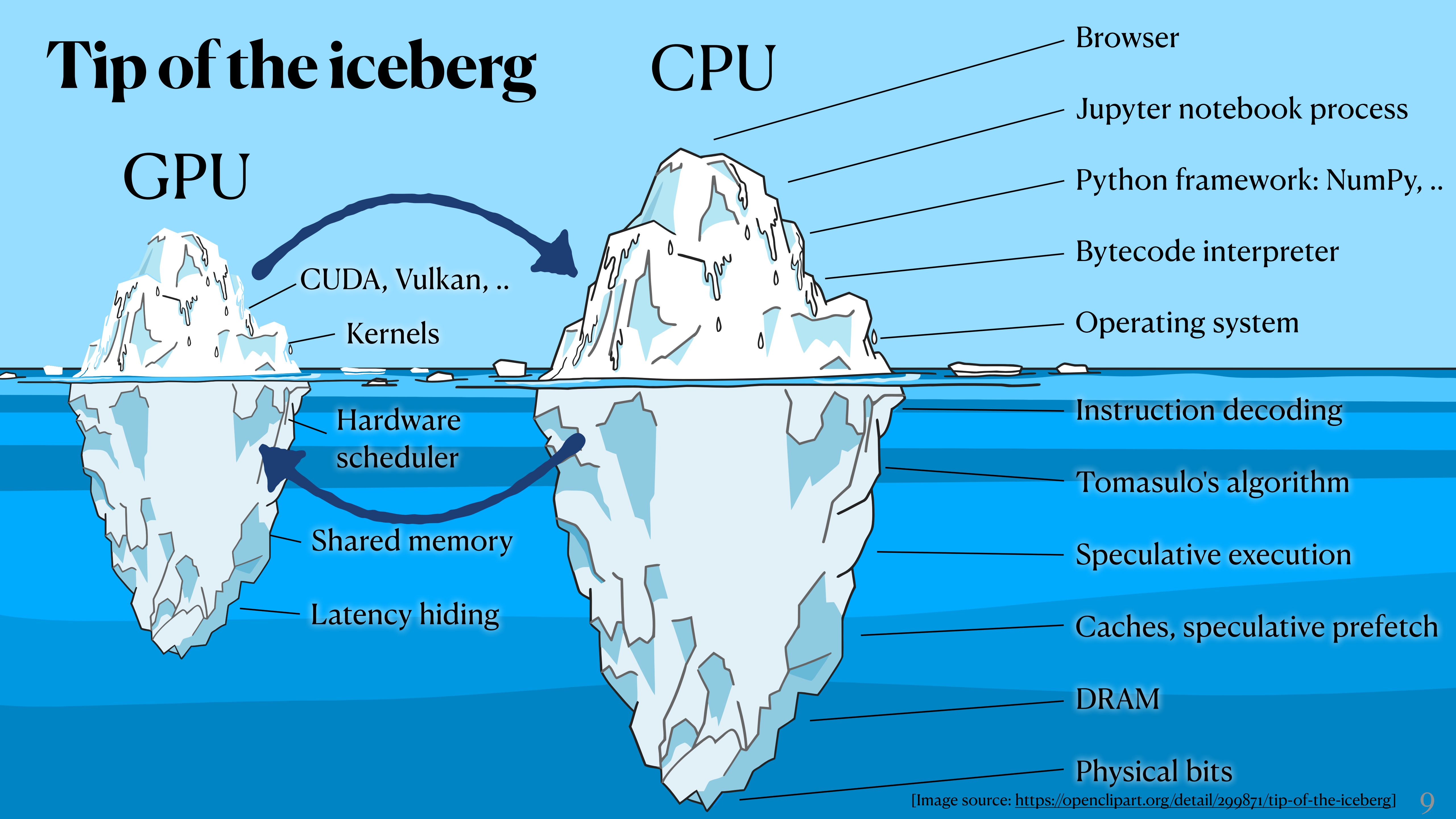
## GPU



# Tip of the iceberg

## CPU

## GPU



Browser

Jupyter notebook process

Python framework: NumPy, ..

Bytecode interpreter

Operating system

CUDA, Vulkan, ..

Kernels

Hardware scheduler

Shared memory

Latency hiding

Instruction decoding

Tomasulo's algorithm

Speculative execution

Caches, speculative prefetch

DRAM

Physical bits

# Model problem

A simple enough operation, but one that isn't "natively" supported by NumPy

- `psum(x)`: sum the positive elements of the input vector "x"

If you want to follow along:

<https://github.com/wjakob/psum>



This repository is *not* relevant for homework or the final exam.

I uploaded it for those interested in the nitty-gritty details.

**Demo time**

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP         44 (to 50)  
6 LOAD_GLOBAL        0 (range)  
8 LOAD_GLOBAL        1 (len)  
10 LOAD_FAST          0 (x)  
12 CALL_FUNCTION      1  
14 CALL_FUNCTION      1  
16 GET_ITER  
>> 18 FOR_ITER           28 (to 48)  
20 STORE_FAST        2 (i)  
  
22 LOAD_FAST          0 (x)  
24 LOAD_FAST          2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST         1 (0)  
30 COMPARE_OP         4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST          1 (r)  
36 LOAD_FAST          0 (x)  
38 LOAD_FAST          2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST         1 (r)  
46 JUMP_ABSOLUTE     18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST          1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST        0 (x)  
12 CALL_FUNCTION    1  
14 CALL_FUNCTION    1  
16 GET_ITER  
>> 18 FOR_ITER        28 (to 48)  
20 STORE_FAST      2 (i)  
  
22 LOAD_FAST       0 (x)  
24 LOAD_FAST       2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST     1 (0)  
30 COMPARE_OP     4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST       1 (r)  
36 LOAD_FAST       0 (x)  
38 LOAD_FAST       2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST     1 (r)  
46 JUMP_ABSOLUTE  18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST       1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST        0 (x)  
12 CALL_FUNCTION     1  
14 CALL_FUNCTION     1  
16 GET_ITER  
>> 18 FOR_ITER          28 (to 48)  
20 STORE_FAST       2 (i)  
  
→ 22 LOAD_FAST        0 (x)  
24 LOAD_FAST        2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST       1 (0)  
30 COMPARE_OP       4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST        1 (r)  
36 LOAD_FAST        0 (x)  
38 LOAD_FAST        2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST       1 (r)  
46 JUMP_ABSOLUTE    18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST        1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

```
0 LOAD_CONST          1 (0)  
2 STORE_FAST          1 (r)  
  
4 SETUP_LOOP          44 (to 50)  
6 LOAD_GLOBAL         0 (range)  
8 LOAD_GLOBAL         1 (len)  
10 LOAD_FAST           0 (x)  
12 CALL_FUNCTION       1  
14 CALL_FUNCTION       1  
16 GET_ITER  
>> 18 FOR_ITER           28 (to 48)  
20 STORE_FAST         2 (i)  
  
→ 22 LOAD_FAST          0 (x)  
24 LOAD_FAST          2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST         1 (0)  
30 COMPARE_OP         4 (>)  
32 POP_JUMP_IF_FALSE  18  
  
34 LOAD_FAST          1 (r)  
36 LOAD_FAST          0 (x)  
38 LOAD_FAST          2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST         1 (r)  
46 JUMP_ABSOLUTE     18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST          1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

Stack

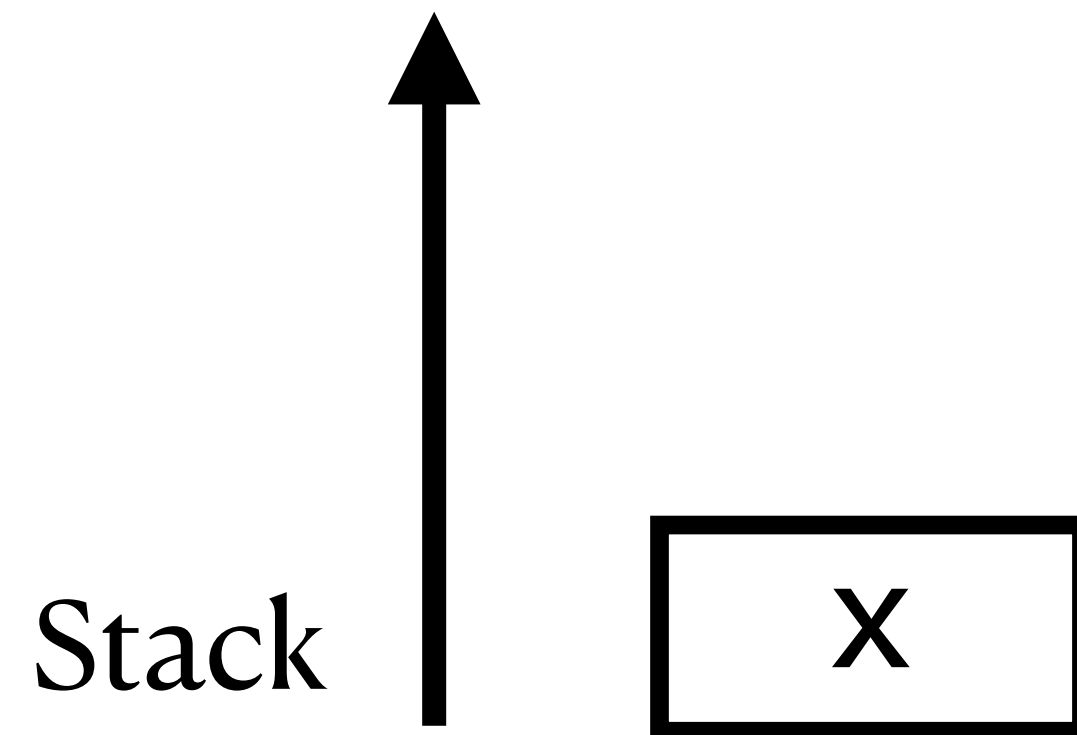


	0	LOAD_CONST	1	(0)
	2	STORE_FAST	1	(r)
	4	SETUP_LOOP	44	(to 50)
	6	LOAD_GLOBAL	0	(range)
	8	LOAD_GLOBAL	1	(len)
	10	LOAD_FAST	0	(x)
	12	CALL_FUNCTION	1	
	14	CALL_FUNCTION	1	
	16	GET_ITER		
>>	18	FOR_ITER	28	(to 48)
	20	STORE_FAST	2	(i)
→	22	LOAD_FAST	0	(x)
	24	LOAD_FAST	2	(i)
	26	BINARY_SUBSCR		
	28	LOAD_CONST	1	(0)
	30	COMPARE_OP	4	(>)
	32	POP_JUMP_IF_FALSE	18	
	34	LOAD_FAST	1	(r)
	36	LOAD_FAST	0	(x)
	38	LOAD_FAST	2	(i)
	40	BINARY_SUBSCR		
	42	INPLACE_ADD		
	44	STORE_FAST	1	(r)
	46	JUMP_ABSOLUTE	18	
>>	48	POP_BLOCK		
>>	50	LOAD_FAST	1	(r)
	52	RETURN_VALUE		

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

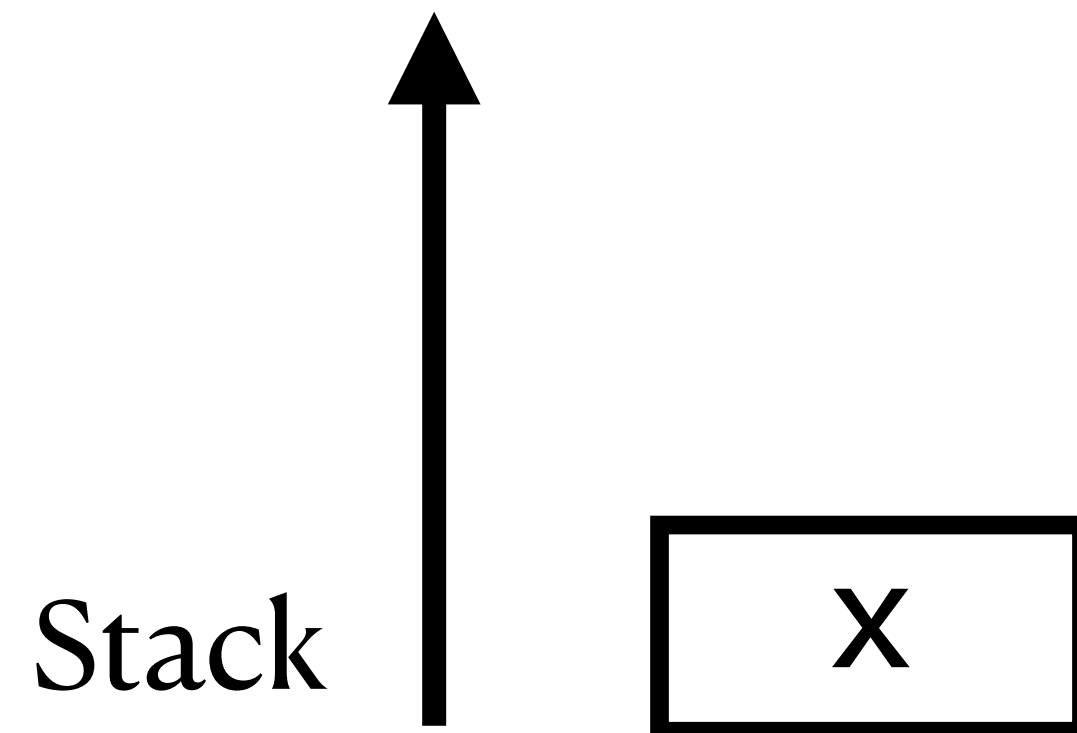


	0	LOAD_CONST	1	(0)
	2	STORE_FAST	1	(r)
	4	SETUP_LOOP	44	(to 50)
	6	LOAD_GLOBAL	0	(range)
	8	LOAD_GLOBAL	1	(len)
	10	LOAD_FAST	0	(x)
	12	CALL_FUNCTION	1	
	14	CALL_FUNCTION	1	
	16	GET_ITER		
>>	18	FOR_ITER	28	(to 48)
	20	STORE_FAST	2	(i)
→	22	LOAD_FAST	0	(x)
	24	LOAD_FAST	2	(i)
	26	BINARY_SUBSCR		
	28	LOAD_CONST	1	(0)
	30	COMPARE_OP	4	(>)
	32	POP_JUMP_IF_FALSE	18	
	34	LOAD_FAST	1	(r)
	36	LOAD_FAST	0	(x)
	38	LOAD_FAST	2	(i)
	40	BINARY_SUBSCR		
	42	INPLACE_ADD		
	44	STORE_FAST	1	(r)
	46	JUMP_ABSOLUTE	18	
>>	48	POP_BLOCK		
>>	50	LOAD_FAST	1	(r)
	52	RETURN_VALUE		

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

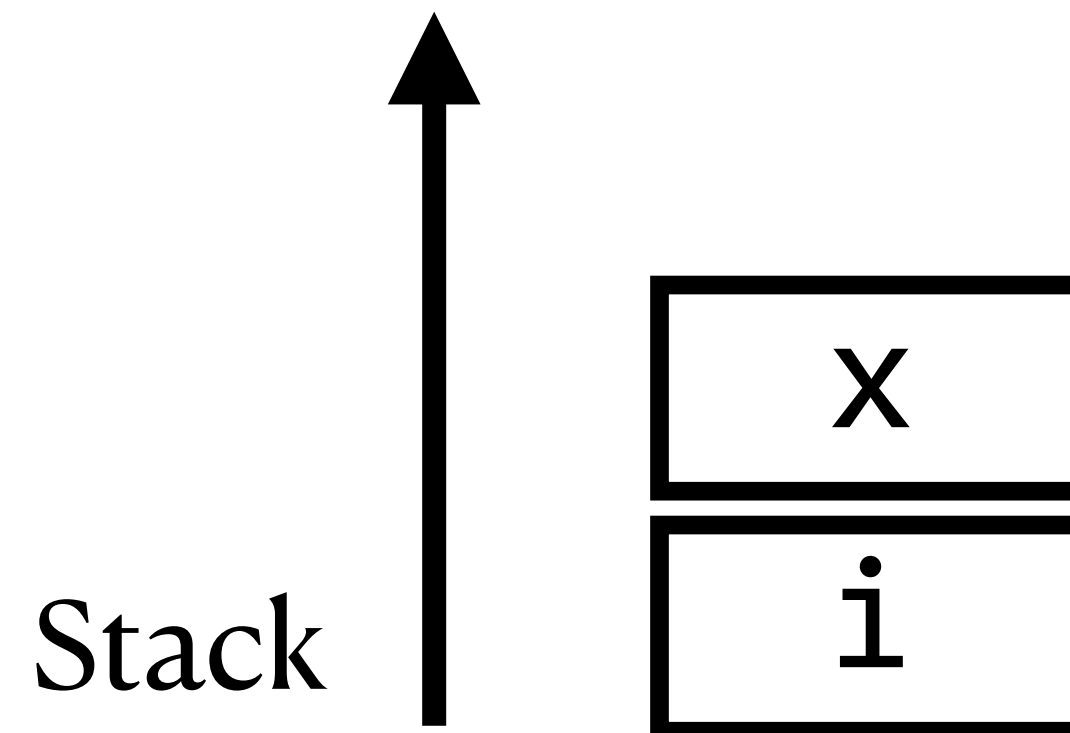


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST        0 (x)  
12 CALL_FUNCTION     1  
14 CALL_FUNCTION     1  
16 GET_ITER  
>> 18 FOR_ITER         28 (to 48)  
20 STORE_FAST       2 (i)  
  
22 LOAD_FAST        0 (x)  
24 LOAD_FAST        2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST       1 (0)  
30 COMPARE_OP       4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST        1 (r)  
36 LOAD_FAST        0 (x)  
38 LOAD_FAST        2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST       1 (r)  
46 JUMP_ABSOLUTE    18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST        1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

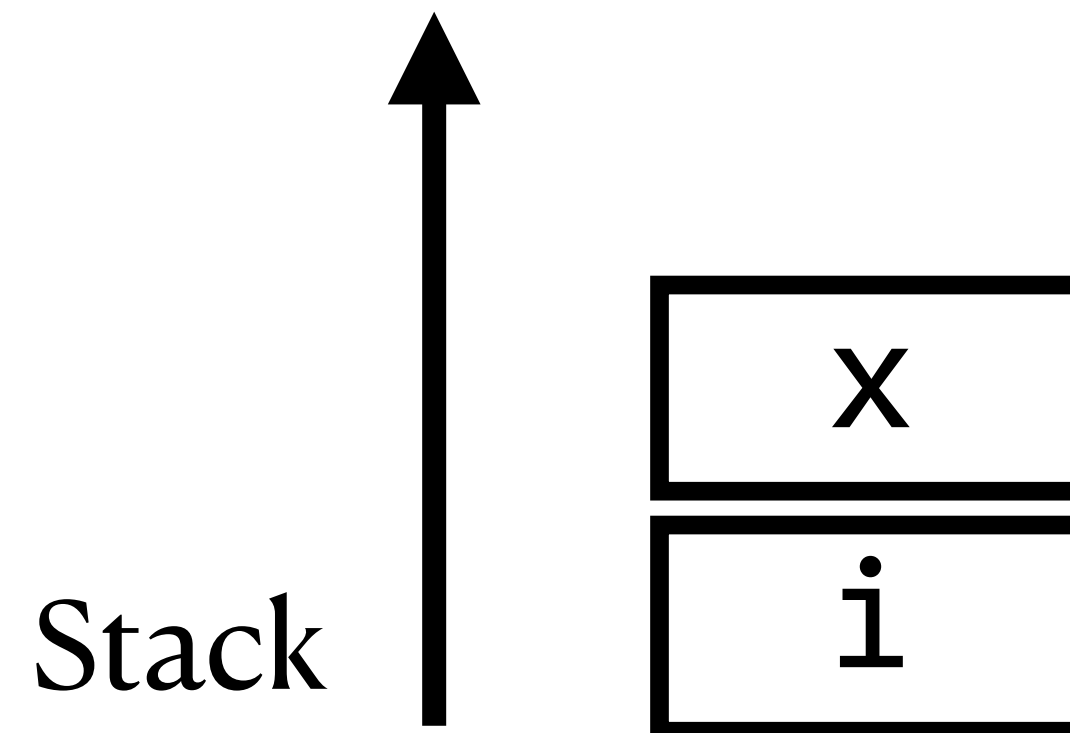


	0	LOAD_CONST	1	(0)
	2	STORE_FAST	1	(r)
	4	SETUP_LOOP	44	(to 50)
	6	LOAD_GLOBAL	0	(range)
	8	LOAD_GLOBAL	1	(len)
	10	LOAD_FAST	0	(x)
	12	CALL_FUNCTION	1	
	14	CALL_FUNCTION	1	
	16	GET_ITER		
>>	18	FOR_ITER	28	(to 48)
	20	STORE_FAST	2	(i)
	22	LOAD_FAST	0	(x)
→	24	LOAD_FAST	2	(i)
	26	BINARY_SUBSCR		
	28	LOAD_CONST	1	(0)
	30	COMPARE_OP	4	(>)
	32	POP_JUMP_IF_FALSE	18	
	34	LOAD_FAST	1	(r)
	36	LOAD_FAST	0	(x)
	38	LOAD_FAST	2	(i)
	40	BINARY_SUBSCR		
	42	INPLACE_ADD		
	44	STORE_FAST	1	(r)
	46	JUMP_ABSOLUTE	18	
>>	48	POP_BLOCK		
>>	50	LOAD_FAST	1	(r)
	52	RETURN_VALUE		

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

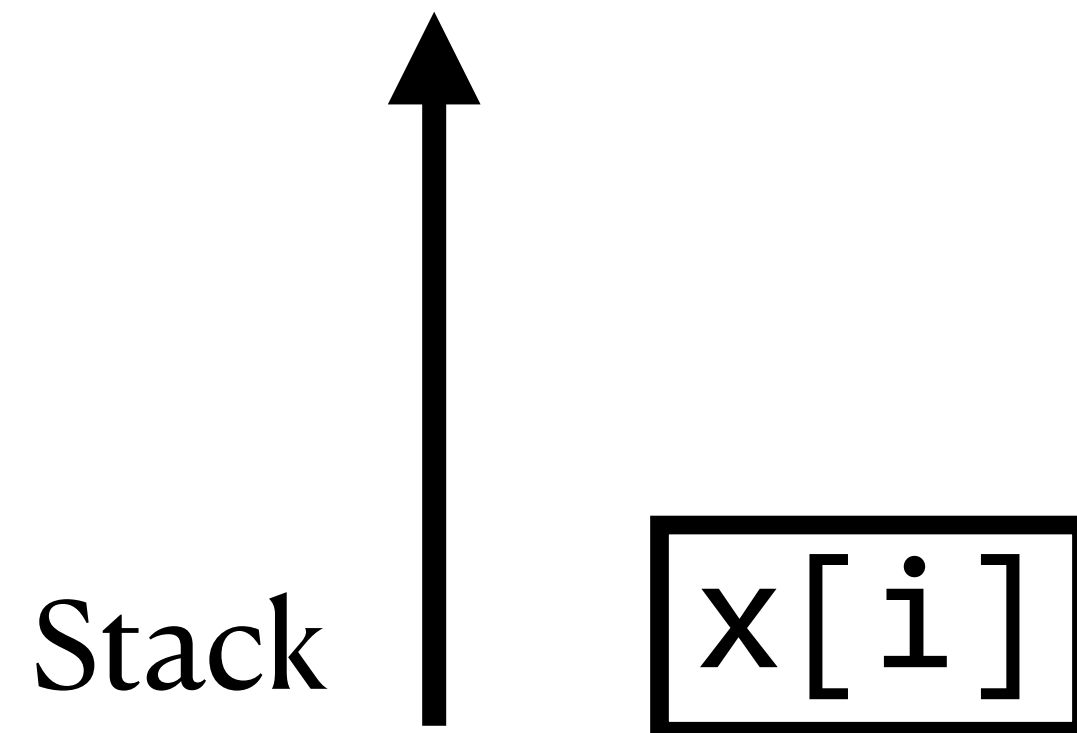


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP         44 (to 50)  
6 LOAD_GLOBAL        0 (range)  
8 LOAD_GLOBAL        1 (len)  
10 LOAD_FAST         0 (x)  
12 CALL_FUNCTION      1  
14 CALL_FUNCTION      1  
16 GET_ITER  
>> 18 FOR_ITER          28 (to 48)  
20 STORE_FAST        2 (i)  
  
22 LOAD_FAST         0 (x)  
24 LOAD_FAST         2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST        1 (0)  
30 COMPARE_OP        4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST         1 (r)  
36 LOAD_FAST         0 (x)  
38 LOAD_FAST         2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST        1 (r)  
46 JUMP_ABSOLUTE     18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST         1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

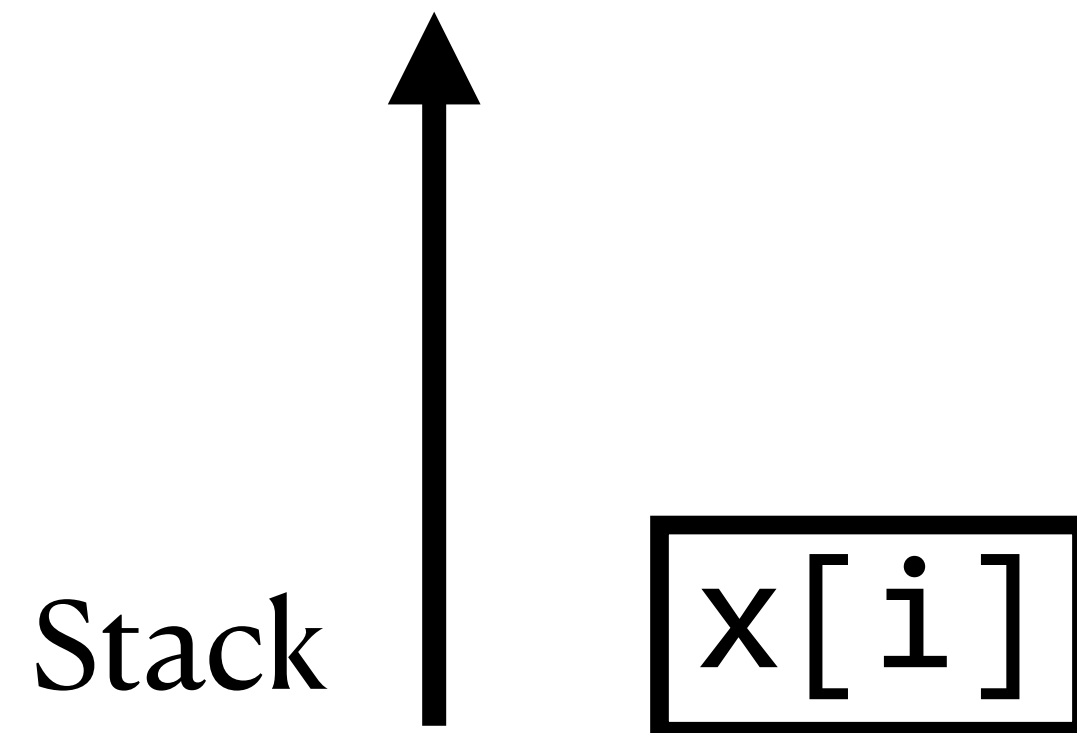


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST          1 (r)  
  
4 SETUP_LOOP          44 (to 50)  
6 LOAD_GLOBAL         0 (range)  
8 LOAD_GLOBAL         1 (len)  
10 LOAD_FAST           0 (x)  
12 CALL_FUNCTION       1  
14 CALL_FUNCTION       1  
16 GET_ITER  
>> 18 FOR_ITER           28 (to 48)  
20 STORE_FAST          2 (i)  
  
22 LOAD_FAST           0 (x)  
24 LOAD_FAST           2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST          1 (0)  
30 COMPARE_OP          4 (>)  
32 POP_JUMP_IF_FALSE  18  
  
34 LOAD_FAST           1 (r)  
36 LOAD_FAST           0 (x)  
38 LOAD_FAST           2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST          1 (r)  
46 JUMP_ABSOLUTE      18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST           1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

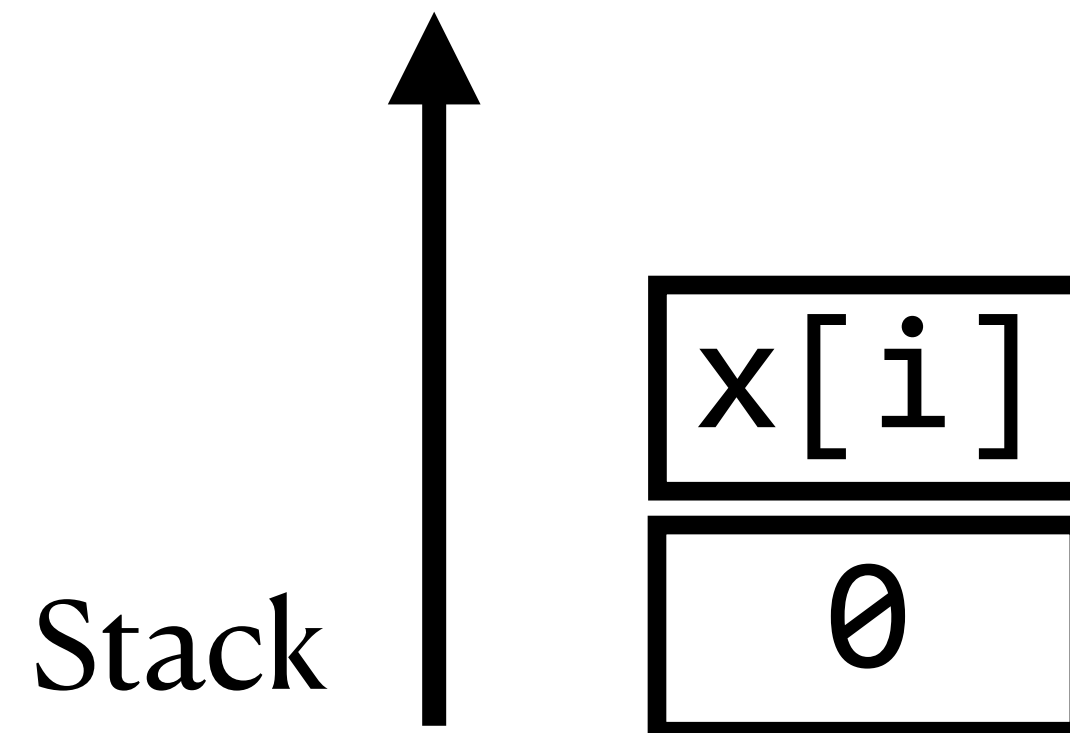


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST        0 (x)  
12 CALL_FUNCTION     1  
14 CALL_FUNCTION     1  
16 GET_ITER  
>> 18 FOR_ITER         28 (to 48)  
20 STORE_FAST       2 (i)  
  
22 LOAD_FAST        0 (x)  
24 LOAD_FAST        2 (i)  
26 BINARY_SUBSCR  
→ 28 LOAD_CONST      1 (0)  
30 COMPARE_OP       4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST        1 (r)  
36 LOAD_FAST        0 (x)  
38 LOAD_FAST        2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST       1 (r)  
46 JUMP_ABSOLUTE    18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST        1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

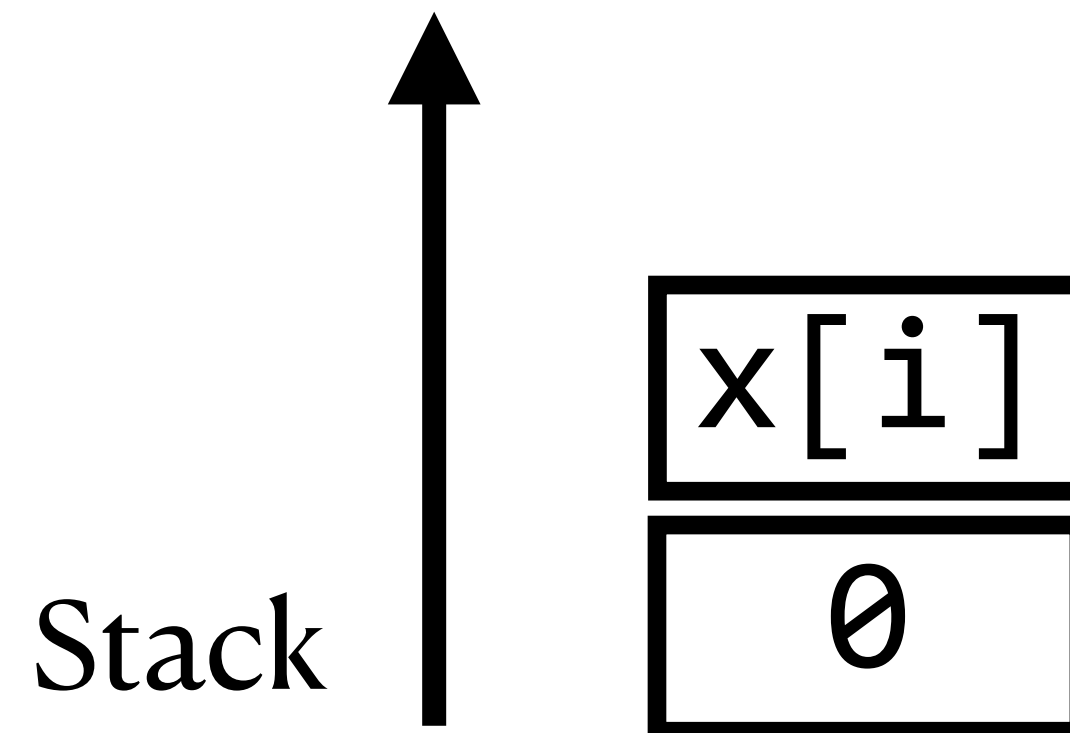


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST        0 (x)  
12 CALL_FUNCTION     1  
14 CALL_FUNCTION     1  
16 GET_ITER  
>> 18 FOR_ITER         28 (to 48)  
20 STORE_FAST       2 (i)  
  
22 LOAD_FAST        0 (x)  
24 LOAD_FAST        2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST       1 (0)  
30 COMPARE_OP       4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST        1 (r)  
36 LOAD_FAST        0 (x)  
38 LOAD_FAST        2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST       1 (r)  
46 JUMP_ABSOLUTE    18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST        1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```

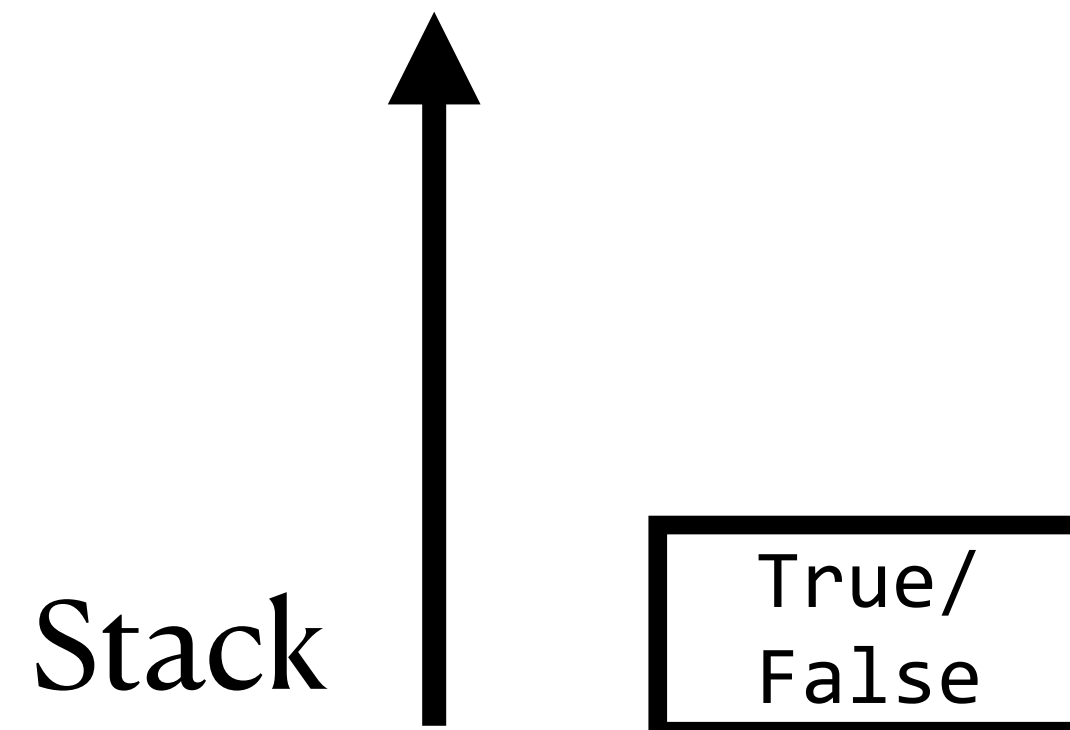


```
0 LOAD_CONST          1 (0)  
2 STORE_FAST         1 (r)  
  
4 SETUP_LOOP        44 (to 50)  
6 LOAD_GLOBAL       0 (range)  
8 LOAD_GLOBAL       1 (len)  
10 LOAD_FAST         0 (x)  
12 CALL_FUNCTION     1  
14 CALL_FUNCTION     1  
16 GET_ITER  
>> 18 FOR_ITER         28 (to 48)  
20 STORE_FAST        2 (i)  
  
22 LOAD_FAST         0 (x)  
24 LOAD_FAST         2 (i)  
26 BINARY_SUBSCR  
28 LOAD_CONST        1 (0)  
30 COMPARE_OP        4 (>)  
32 POP_JUMP_IF_FALSE 18  
  
34 LOAD_FAST         1 (r)  
36 LOAD_FAST         0 (x)  
38 LOAD_FAST         2 (i)  
40 BINARY_SUBSCR  
42 INPLACE_ADD  
44 STORE_FAST        1 (r)  
46 JUMP_ABSOLUTE    18  
>> 48 POP_BLOCK  
  
>> 50 LOAD_FAST         1 (r)  
52 RETURN_VALUE
```

# Python bytecode

```
def psum(x):  
    r = 0  
    for i in range(len(x)):  
        if x[i] > 0:  
            r += x[i]  
    return r
```

```
# Bytecode disassembler  
import dis  
dis.dis(psum)
```



	0	LOAD_CONST	1	(0)
	2	STORE_FAST	1	(r)
	4	SETUP_LOOP	44	(to 50)
	6	LOAD_GLOBAL	0	(range)
	8	LOAD_GLOBAL	1	(len)
	10	LOAD_FAST	0	(x)
	12	CALL_FUNCTION	1	
	14	CALL_FUNCTION	1	
	16	GET_ITER		
>>	18	FOR_ITER	28	(to 48)
	20	STORE_FAST	2	(i)
	22	LOAD_FAST	0	(x)
	24	LOAD_FAST	2	(i)
	26	BINARY_SUBSCR		
	28	LOAD_CONST	1	(0)
→	30	COMPARE_OP	4	(>)
	32	POP_JUMP_IF_FALSE	18	
	34	LOAD_FAST	1	(r)
	36	LOAD_FAST	0	(x)
	38	LOAD_FAST	2	(i)
	40	BINARY_SUBSCR		
	42	INPLACE_ADD		
	44	STORE_FAST	1	(r)
	46	JUMP_ABSOLUTE	18	
>>	48	POP_BLOCK		
>>	50	LOAD_FAST	1	(r)
	52	RETURN_VALUE		

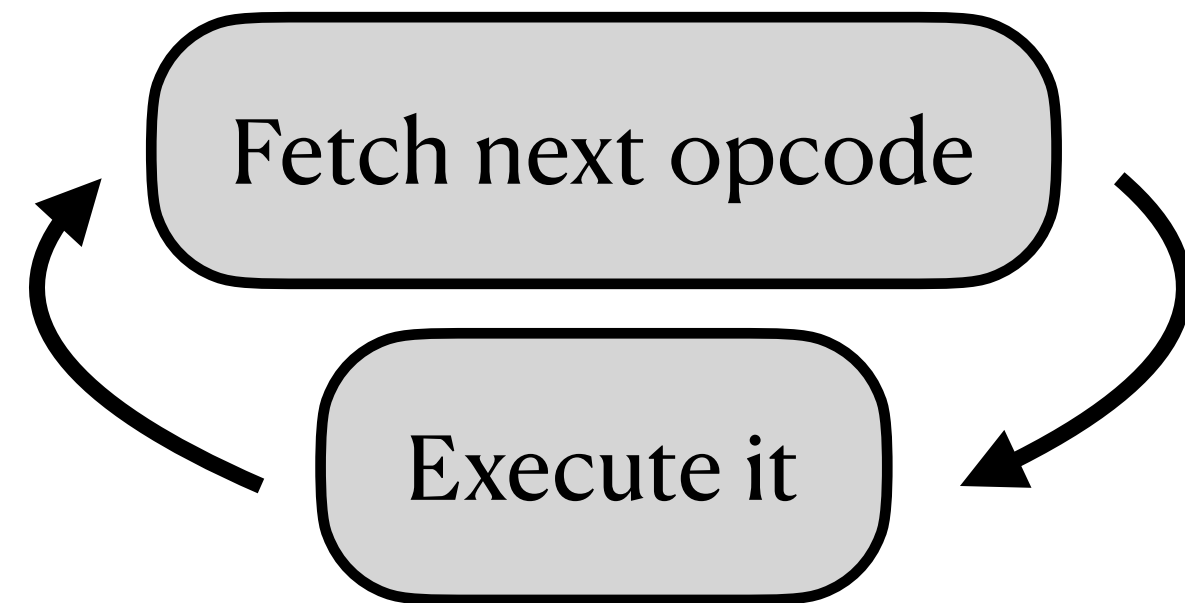
# Python bytecode

	0	LOAD_CONST	1	(0)
	2	STORE_FAST	1	(r)
	4	SETUP_LOOP	44	(to 50)
	6	LOAD_GLOBAL	0	(range)
	8	LOAD_GLOBAL	1	(len)
	10	LOAD_FAST	0	(x)
	12	CALL_FUNCTION	1	
	14	CALL_FUNCTION	1	
	16	GET_ITER		
>>	18	FOR_ITER	28	(to 48)
	20	STORE_FAST	2	(i)
	22	LOAD_FAST	0	(x)
	24	LOAD_FAST	2	(i)
	26	BINARY_SUBSCR		
	28	LOAD_CONST	1	(0)
→	30	COMPARE_OP	4	(>)
	32	POP_JUMP_IF_FALSE	18	
	34	LOAD_FAST	1	(r)
	36	LOAD_FAST	0	(x)
	38	LOAD_FAST	2	(i)
	40	BINARY_SUBSCR		
	42	INPLACE_ADD		
	44	STORE_FAST	1	(r)
	46	JUMP_ABSOLUTE	18	
>>	48	POP_BLOCK		
>>	50	LOAD_FAST	1	(r)
	52	RETURN_VALUE		

# Dissecting the Python interpreter

Let's follow the path of a single instruction

```
switch (opcode) { // ceval.c:1848  
    case ...:  
    case ...:  
    case ...:  
    case ...:  
    case ...:  
    case ...:  
    case ...:  
    case ...:  
}
```

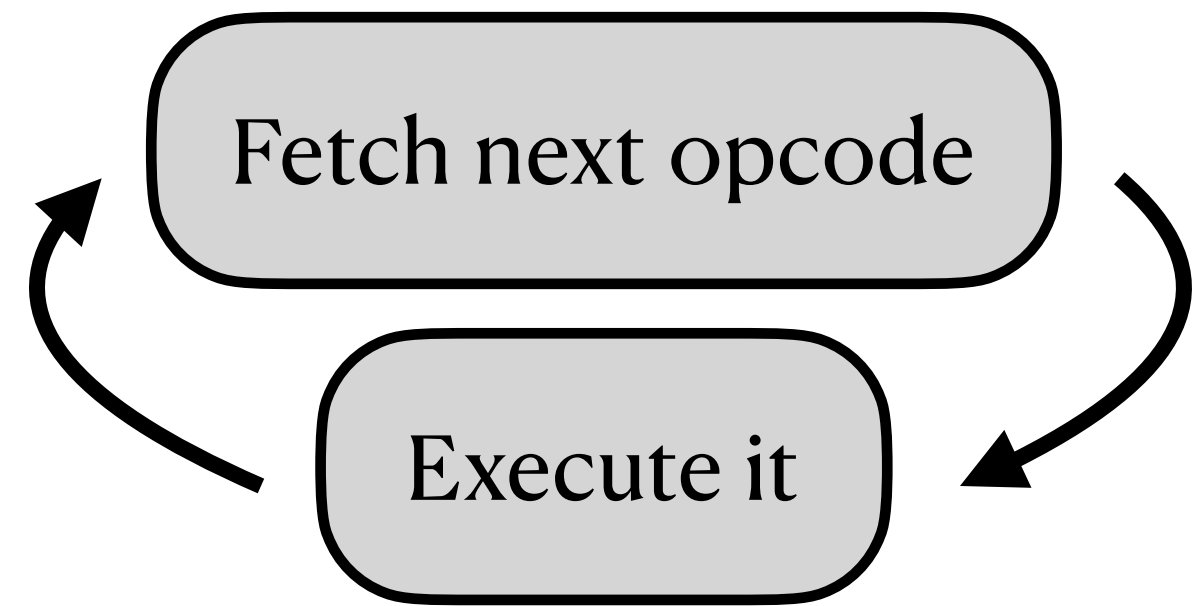


Big loop that fetches the next *opcode* and jumps to one of ~130 different code blocks.

# Dissecting the Python interpreter

Let's follow the path of a single instruction

```
switch (opcode) { // ceval.c:1848
  case ...:
  case ...:
  case ...:
  case ...:
  case ...:
  case ...:
  case ...:
}
```



Big loop that fetches the next *opcode* and jumps to one of ~130 different code blocks.

```
case TARGET(COMPARE_OP): { ceval.c:3629
  assert(oparg <= Py_GE);
  PyObject *right = POP();
  PyObject *left = TOP();
  PyObject *res = PyObject_RichCompare(
    left, right, oparg);
  SET_TOP(res);
  Py_DECREF(left);
  Py_DECREF(right);
  if (res == NULL)
    goto error;
  PREDICT(POP_JUMP_IF_FALSE);
  PREDICT(POP_JUMP_IF_TRUE);
  DISPATCH();
}
```

Pops two operand from stack, pushes result back.  
Reference counting, error management.  
left, right could be anything at this point.

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
case TARGET(COMPARE_OP): { ceval.c:3629
    assert(oparg <= Py_GE);
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyObject_RichCompare(
        left, right, oparg);
    SET_TOP(res);
    Py_DECREF(left);
    Py_DECREF(right);
    if (res == NULL)
        goto error;
    PREDICT(POP_JUMP_IF_FALSE);
    PREDICT(POP_JUMP_IF_TRUE);
    DISPATCH();
}
```

Pops two operand from stack, pushes result back.

Reference counting, error management.

left, right could be anything at this point.

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
case TARGET(COMPARE_OP): { ceval.c:3629
    assert(oparg <= Py_GE);
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyObject_RichCompare(
        left, right, oparg);
    SET_TOP(res);
    Py_DECREF(left);
    Py_DECREF(right);
    if (res == NULL)
        goto error;
    PREDICT(POP_JUMP_IF_FALSE);
    PREDICT(POP_JUMP_IF_TRUE);
    DISPATCH();
}
```



Pops two operand from stack, pushes result back.  
Reference counting, error management.  
left, right could be anything at this point.

```
PyObject * PyObject_RichCompare( // object.c:729
    PyObject *v, PyObject *w, int op) {
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
    if (v == NULL || w == NULL) {
        if (!_PyErr_Occurred(tstate)) {
            PyErr_BadInternalCall();
        }
        return NULL;
    }
    if (_Py_EnterRecursiveCall(tstate,
        " in comparison")) {
        return NULL;
    }
    PyObject *res = do_richcompare(
        tstate, v, w, op);
    _Py_LeaveRecursiveCall(tstate);
    return res;
}
```

Makes sure that there is no stack overflow.

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
PyObject * PyObject_RichCompare( // object.c:729
    PyObject *v, PyObject *w, int op) {
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
    if (v == NULL || w == NULL) {
        if (!_PyErr_Occurred(tstate)) {
            PyErr_BadInternalCall();
        }
        return NULL;
    }
    if (_Py_EnterRecursiveCall(tstate,
        " in comparison")) {
        return NULL;
    }
    PyObject *res = do_richcompare(
        tstate, v, w, op);
    _Py_LeaveRecursiveCall(tstate);
    return res;
}
```

Makes sure that there is no stack overflow.

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
PyObject * PyObject_RichCompare( // object.c:729
    PyObject *v, PyObject *w, int op) {
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
    if (v == NULL || w == NULL) {
        if (!_PyErr_Occurred(tstate)) {
            PyErr_BadInternalCall();
        }
        return NULL;
    }
    if (_Py_EnterRecursiveCall(tstate,
        " in comparison")) {
        return NULL;
    }
    PyObject *res = do_richcompare(
        tstate, v, w, op);
    _Py_LeaveRecursiveCall(tstate);
    return res;
}
```

Makes sure that there is no stack overflow.

```
// object.c:676
static PyObject *
do_richcompare(PyThreadState *tstate,
    PyObject *v, PyObject *w, int op)
{
    richcmpfunc f;
    PyObject *res;
    int checked_reverse_op = 0;

    if (!Py_IS_TYPE(v, Py_TYPE(w)) &&
        PyType_IsSubtype(Py_TYPE(w), Py_TYPE(v)) &&
        (f = Py_TYPE(w)->tp_richcompare) != NULL) {
        checked_reverse_op = 1;
        res = (*f)(w, v, _Py_SwappedOp[op]);
        if (res != Py_NotImplemented)
            return res;
        Py_DECREF(res);
    }
    if ((f = Py_TYPE(v)->tp_richcompare) != NULL) {
        res = (*f)(v, w, op);
        if (res != Py_NotImplemented)
            return res;
        Py_DECREF(res);
    }
    if (!checked_reverse_op &&
        (f = Py_TYPE(w)->tp_richcompare) != NULL) {
        res = (*f)(w, v, _Py_SwappedOp[op]);
        if (res != Py_NotImplemented)
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
PyObject * PyObject_RichCompare( // object.c:729
    PyObject *v, PyObject *w, int op) {
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
    if (v == NULL || w == NULL) {
        if (!_PyErr_Occurred(tstate)) {
            PyErr_BadInternalCall();
        }
        return NULL;
    }
    if (_Py_EnterRecursiveCall(tstate,
        " in comparison")) {
        return NULL;
    }
    PyObject *res = do_richcompare(
        tstate, v, w, op);
    _Py_LeaveRecursiveCall(tstate);
    return res;
}
```

Makes sure that there is no stack overflow.

```
    res = (*f)(v, w, op);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
if (!checked_reverse_op &&
    (f = Py_TYPE(w)->tp_richcompare) != NULL) {
    res = (*f)(w, v, _Py_SwappedOp[op]);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
/* If neither object implements it, provide a sensible default
   for == and !=, but raise an exception for ordering. */
switch (op) {
case Py_EQ:
    res = (v == w) ? Py_True : Py_False;
    break;
case Py_NE:
    res = (v != w) ? Py_True : Py_False;
    break;
default:
    _PyErr_Format(tstate, PyExc_TypeError,
        "'%s' not supported between "
        "instances of '%.100s' and '%.100s'",
        opstrings[op],
        Py_TYPE(v)->tp_name,
        Py_TYPE(w)->tp_name);
}
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
    res = (*f)(v, w, op);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
if (!checked_reverse_op &&
    (f = Py_TYPE(w)->tp_richcompare) != NULL) {
    res = (*f)(w, v, _Py_SwappedOp[op]);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
/* If neither object implements it, provide a sensible default
   for == and !=, but raise an exception for ordering. */
switch (op) {
case Py_EQ:
    res = (v == w) ? Py_True : Py_False;
    break;
case Py_NE:
    res = (v != w) ? Py_True : Py_False;
    break;
default:
    _PyErr_Format(tstate, PyExc_TypeError,
        "'%s' not supported between "
        "instances of '%.100s' and '%.100s'",
        opstrings[op],
        Py_TYPE(v)->tp_name,
        Py_TYPE(w)->tp_name);
}
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

```
    res = (*f)(v, w, op);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
if (!checked_reverse_op &&
    (f = Py_TYPE(w)->tp_richcompare) != NULL) {
    res = (*f)(w, v, _Py_SwappedOp[op]);
    if (res != Py_NotImplemented)
        return res;
    Py_DECREF(res);
}
/* If neither object implements it, provide a sensible default
   for == and !=, but raise an exception for ordering. */
switch (op) {
case Py_EQ:
    res = (v == w) ? Py_True : Py_False;
    break;
case Py_NE:
    res = (v != w) ? Py_True : Py_False;
    break;
default:
    _PyErr_Format(tstate, PyExc_TypeError,
                  "'%s' not supported between "
                  "instances of '%.100s' and '%.100s'",
                  opstrings[op],
                  Py_TYPE(v)->tp_name,
                  Py_TYPE(w)->tp_name);
}
```



[floatobject.c:362](#)

```
/* Comparison is pretty much a nightmare. When comparing
 * we do it as straightforwardly (and long-windedly) as co
 * that, e.g., Python x == y delivers the same result as t
 * C x == y when x and/or y is a NaN.
 * When mixing float with an integer type, there's no good
 * Converting the double to an integer obviously doesn't w
 * may lose info from fractional bits. Converting the int
 * also has two failure modes: (1) an int may trigger ove
 * large to fit in the dynamic range of a C double); (2) e
 * more bits than fit in a C double (e.g., on a 64-bit box
 * 63 bits of precision, but a C double probably has only
 * we can falsely claim equality when low-order integer bi
 * coercion to double. So this part is painful too.
 */
```

```
static PyObject*
float_richcompare(PyObject *v, PyObject *w, int op)
{
    double i, j;
    int r = 0;

    assert(PyFloat_Check(v));
    i = PyFloat_AS_DOUBLE(v);

    /* Switch on the type of w. Set i and j to doubles to
     * and on to the richcompare to use
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

[floatobject.c:362](#)

```
/* Comparison is pretty much a nightmare. When comparing float to float,
 * we do it as straightforwardly (and long-windedly) as conceivable, so
 * that, e.g., Python x == y delivers the same result as the platform
 * C x == y when x and/or y is a NaN.
 * When mixing float with an integer type, there's no good *uniform* approach.
 * Converting the double to an integer obviously doesn't work, since we
 * may lose info from fractional bits. Converting the integer to a double
 * also has two failure modes: (1) an int may trigger overflow (too
 * large to fit in the dynamic range of a C double); (2) even a C long may have
 * more bits than fit in a C double (e.g., on a 64-bit box long may have
 * 63 bits of precision, but a C double probably has only 53), and then
 * we can falsely claim equality when low-order integer bits are lost by
 * coercion to double. So this part is painful too.
 */
```

```
static PyObject*
float_richcompare(PyObject *v, PyObject *w, int op)
{
    double i, j;
    int r = 0;

    assert(PyFloat_Check(v));
    i = PyFloat_AS_DOUBLE(v);

    /* Switch on the type of w. Set i and j to doubles to be compared,
     * and on to the richcompare function.
     */
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

`goto UNIMPLEMENTED;`

Compare:

```
switch (op) {
case Py_EQ:
    r = i == j;
    break;
case Py_NE:
    r = i != j;
    break;
case Py_LE:
    r = i <= j;
    break;
case Py_GE:
    r = i >= j;
    break;
case Py_LT:
    r = i < j;
    break;
case Py_GT:
    r = i > j;
    break;
}
return PyBool_FromLong(r);
```

Unimplemented:

```
Py_RETURN_NOTIMPLEMENTED;
}
```

# Dissecting the Python interpreter

## Let's follow the path of a single instruction

`goto UNIMPLEMENTED;`

Compare:

```
switch (op) {  
  case Py_EQ:  
    r = i == j;  
    break;  
  case Py_NE:  
    r = i != j;  
    break;  
  case Py_LE:  
    r = i <= j;  
    break;  
  case Py_GE:  
    r = i >= j;  
    break;  
  case Py_LT:  
    r = i < j;  
    break;  
  case Py_GT:  
    r = i > j;  
    break;  
}  
return PyBool_FromLong(r);
```

Unimplemented:

```
Py_RETURN_NOTIMPLEMENTED;  
}
```

# Let's take a deep breath..

- That's *a lot* of code for something so simple.



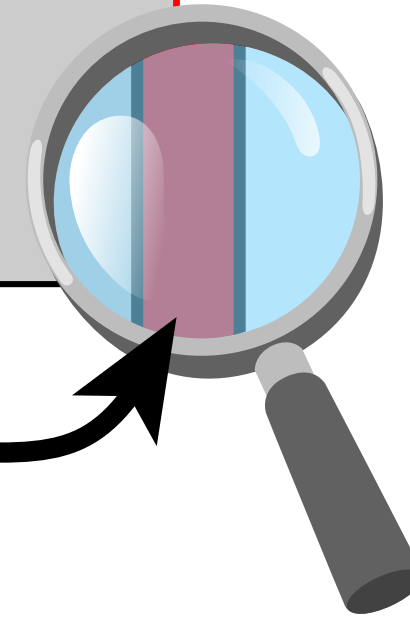
**MidJourney:** *a group of university students in a lecture hall taking a deep breath*

# Let's take a deep breath..

- That's *a lot* of code for something so simple.

Overhead

Actual operation



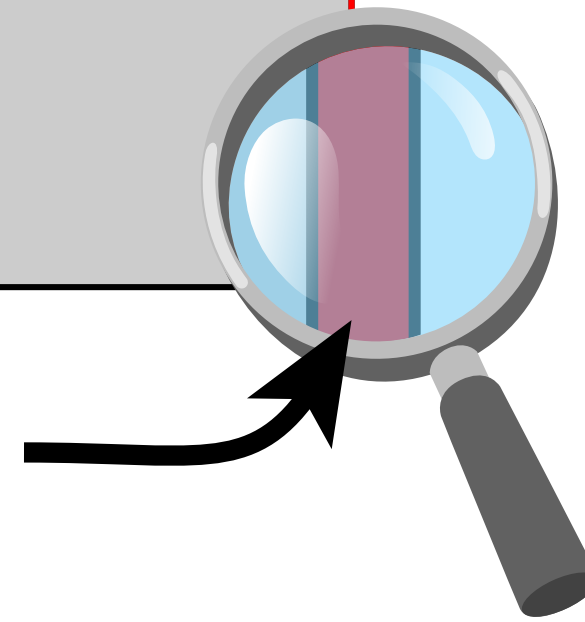
**MidJourney:** *a group of university students in a lecture hall taking a deep breath*

# Let's take a deep breath..

- That's *a lot* of code for something so simple.

Overhead

Actual operation



- Inherent compromise of an interpreted language.
- Overheads usually negligible when the operation does more work.



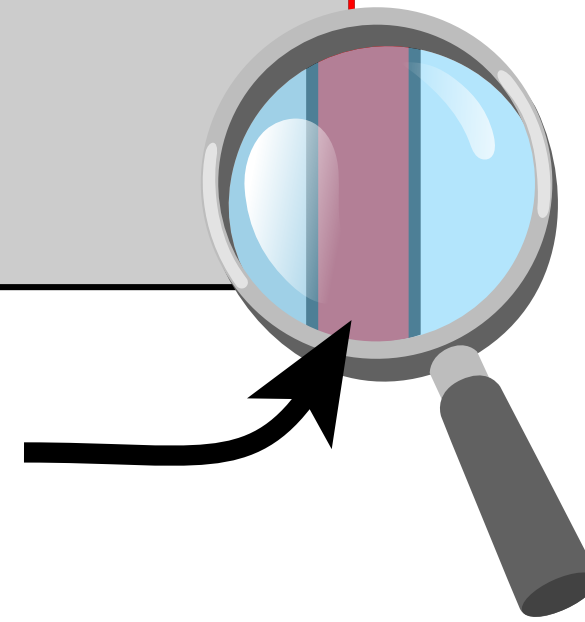
**MidJourney:** *a group of university students in a lecture hall taking a deep breath*

# Let's take a deep breath..

- That's *a lot* of code for something so simple.

Overhead

Actual operation



- Inherent compromise of an interpreted language.
- Overheads usually negligible when the operation does more work.
- Let's try to get closer to the metal (*silicon*, in this case).



**MidJourney:** *a group of university students in a lecture hall taking a deep breath*

# First version in C++

```
float psum_0(float *x, size_t size) {  
    float r { 0.f };  
  
    for (size_t i = 0; i < size; ++i) {  
        if (x[i] >= 0) {  
            r += x[i];  
            if (std::isinf(r))  
                abort();  
        }  
    }  
  
    return r;  
}
```

Everything has types!

float: single precision FP value

size\_t: 64-bit integer

std::isinf(x) – is "x" infinity?

abort(): terminate the program.

float\* x: pointer to a float array



The diagram shows a white arrow pointing from the text "float\* x: pointer to a float array" to a set of curly braces containing the elements "x[0], x[1], x[2], ..." representing a float array.

# Python bindings

## Shamesless advertisement

- The details here aren't super-important.
- Need to "glue" C++ and Python code together, which is done using the nanobind library developed by me.

```
#include <nanobind/ndarray.h>
```

```
NB_MODULE(psum, m) {  
    namespace nb = nanobind;
```

```
    using Array = nb::ndarray<float, nb::shape<nb::any>, nb::c_contig>;
```

```
    m.def("psum_0", [](Array a) { return psum_0(a.data(), a.shape(0)); })  
        .def("psum_1", [](Array a) { return psum_1(a.data(), a.shape(0)); })  
        .def("psum_2", [](Array a) { return psum_2(a.data(), a.shape(0)); })  
        .def("psum_3", [](Array a) { return psum_3(a.data(), a.shape(0)); });
```

```
}
```



<https://nanobind.readthedocs.io/en/latest/>

**Demo time**

# Meaning of "vectorization" in Python

We will shortly see another meaning of that word as well..

Bad:

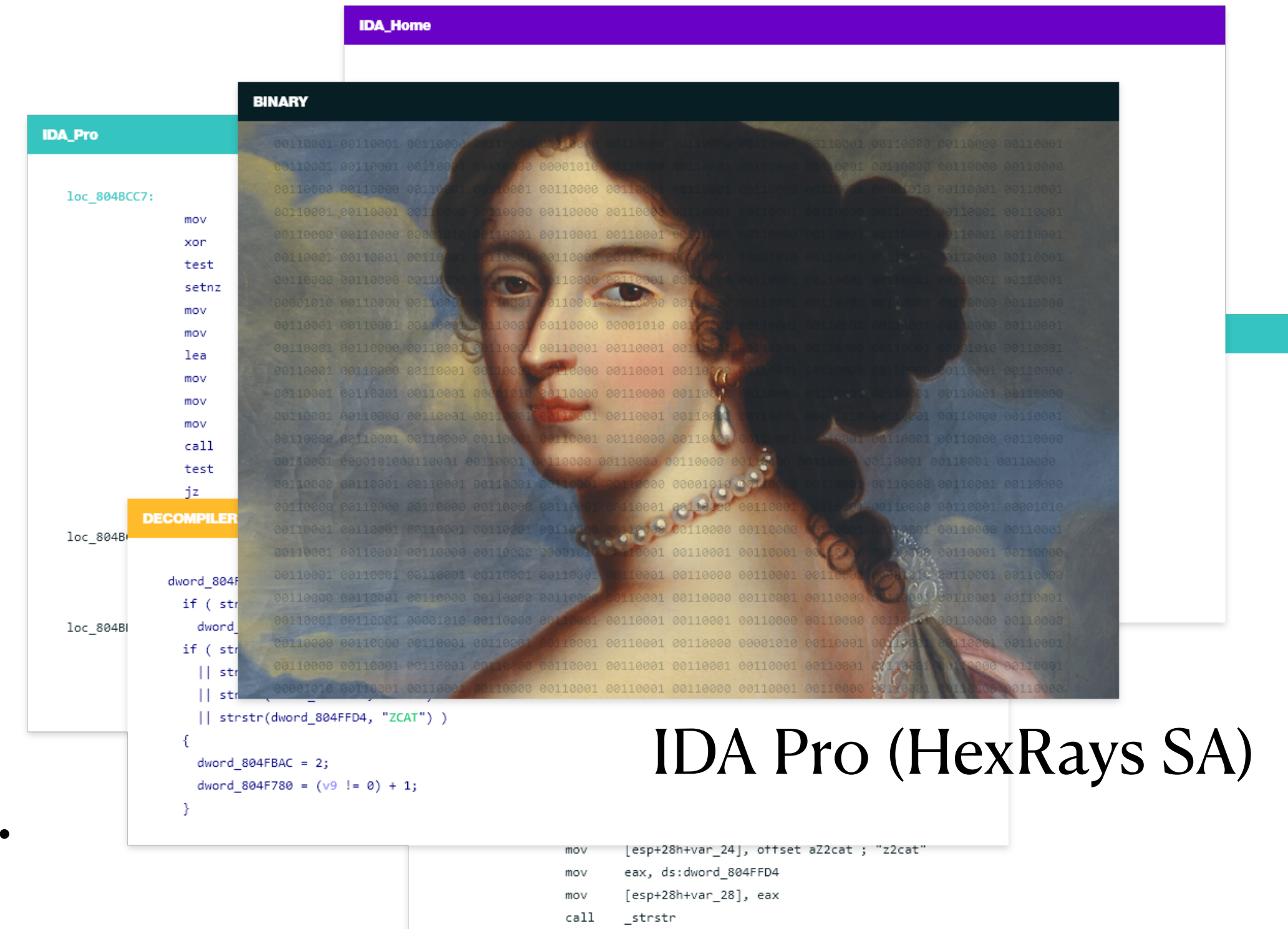
```
for i in range(len(A)):
    if B[i] < 0:
        X[i] = A[i] + C[i]
    else:
        X[i] = A[i] + D[i]
```

Good:

```
X = A + np.where(B < 0, C, D)
```

# A Disassembler

- Can reveal the processor instructions ("assembly code") in a compiled executable
- Often used for hacking/security analysis.
- We will use it to understand the performance differences between the different **psum** variations.



IDA Pro (HexRays SA)

- Freely available: **IDA Home** (<https://hex-rays.com/ida-free/>, Intel processors only),  
Open source alternative: **Ghidra** (<https://ghidra-sre.org/>)

**Demo time**

# Disassembly of psum\_0 – in flat form

```
; ===== S U B R O U T I N E =====
; Attributes: bp-based frame
; __int64 __fastcall psum_0(float *, unsigned __int64)
EXPORT __Z6psum_0Pfm ; CODE XREF: sub_3968+5C↓p
__Z6psum_0Pfm
var_s0 = 0
FD 7B BF A9 STP X29, X30, [SP, #-0x10+var_s0]! ; Store Pair
FD 03 00 91 MOV X29, SP ; Rd = Op2
00 E4 00 2F MOVI D0, #0 ; Move Immediate
A1 01 00 B4 CBZ X1, loc_36B8 ; Compare and Branch on Zero
08 F0 AF 52 MOV W8, #0x7F800000 ; Rd = Op2

loc_368C ; CODE XREF: psum_0(float *,ulong)+3C↓j
01 00 40 BD LDR S1, [X0] ; Load from Memory
28 20 20 1E FCMP S1, #0.0 ; Floating-point Compare
CB 00 00 54 B.LT loc_36AC ; Branch
00 28 21 1E FADD S0, S0, S1 ; Floating-point Add
01 C0 20 1E FABS S1, S0 ; Floating-point Absolute Value
02 01 27 1E FMOV S2, W8 ; Floating-point Move
20 20 22 1E FCMP S1, S2 ; Floating-point Compare
C0 00 00 54 B.EQ loc_36C0 ; Branch

loc_36AC ; CODE XREF: psum_0(float *,ulong)+1C↑j
00 10 00 91 ADD X0, X0, #4 ; Rd = Op1 + Op2
21 04 00 F1 SUBS X1, X1, #1 ; Rd = Op1 - Op2
C1 FE FF 54 B.NE loc_368C ; Branch

loc_36B8 ; CODE XREF: psum_0(float *,ulong)+C↑j
FD 7B C1 A8 LDP X29, X30, [SP+var_s0], #0x10 ; Load Pair
C0 03 5F D6 RET ; Return from Subroutine
; -----

loc_36C0 ; CODE XREF: psum_0(float *,ulong)+30↑j
BB 01 00 94 BL __abort ; Branch with Link
; End of function psum_0(float *,ulong)

; ===== S U B R O U T I N E =====
```

# Disassembly of psum\_0 – in graph form

```
; Segment type: Pure code
AREA __text, CODE
; ORG 0x3678
CODE64

; Attributes: bp-based frame

; __int64 __fastcall psum_0(float *, unsigned __int64)
EXPORT __Z6psum_0Pfm
__Z6psum_0Pfm

var_s0= 0

STP      X29, X30, [SP,#-0x10+var_s0]! ; Store Pair
MOV      X29, SP ; Rd = Op2
MOVI     D0, #0 ; Move Immediate
CBZ      X1, loc_36B8 ; Compare and Branch on Zero
```

```
MOV      W8, #0x7F800000 ; Rd = Op2
```

```
loc_368C
LDR      S1, [X0] ; Load from Memory
FCMP     S1, #0.0 ; Floating-point Compare
B.LT     loc_36AC ; Branch
```

```
FADD     S0, S0, S1 ; Floating-point Add
FABS     S1, S0 ; Floating-point Absolute Value
FMOV     S2, W8 ; Floating-point Move
FCMP     S1, S2 ; Floating-point Compare
B.EQ     loc_36C0 ; Branch
```

```
loc_36AC
ADD      X0, X0, #4 ; Rd = Op1 + Op2
SUBS     X1, X1, #1 ; Rd = Op1 - Op2
B.NE     loc_368C ; Branch
```

```
loc_36C0
BL       _abort ; Branch with Link
; End of function psum_0(float *,ulong)
```

```
loc_36B8
LDP      X29, X30, [SP+var_s0],#0x10 ; Load Pair
RET
```

# Disassembly of psum\_0 – in graph form

```
loc_368C
LDR      S1, [X0] ; Load from Memory
FCMP    S1, #0.0 ; Floating-point Compare
B.LT    loc_36AC ; Branch
```

```
FADD    S0, S0, S1 ; Floating-point Add
FABS    S1, S0 ; Floating-point Absolute Value
FMOV    S2, W8 ; Floating-point Move
FCMP    S1, S2 ; Floating-point Compare
B.EQ    loc_36C0 ; Branch
```

```
loc_36AC
ADD     X0, X0, #4 ; Rd = Op1 + Op2
SUBS    X1, X1, #1 ; Rd = Op1 - Op2
B.NE    loc_368C ; Branch
```

```
loc_36C0
BL      _abort ; Branch with Link
; End of function psum_0(float *,ulong)
```

# Disassembly of psum\_0 – in graph form

```
loc_368C
LDR      S1, [X0] ; Load from Memory
FCMP    S1, #0.0 ; Floating-point Compare
B.LT    loc_36AC ; Branch
```

```
FADD    S0, S0, S1 ; Floating-point Add
FABS    S1, S0 ; Floating-point Absolute Value
FMOV    S2, W8 ; Floating-point Move
FCMP    S1, S2 ; Floating-point Compare
B.EQ    loc_36C0 ; Branch
```

```
loc_36AC
ADD     X0, X0, #4 ; Rd = Op1 + Op2
SUBS    X1, X1, #1 ; Rd = Op1 - Op2
B.NE    loc_368C ; Branch
```

```
loc_36C0
BL     _abort ; Branch with Link
; End of function psum_0(float *,ulong)
```

# A real-life analogy: laundry day!

- It's time to do the laundry: we have 3 batches
  - **Dark**, **Light**, **Color**

**Dark**



**Light**



**Color**



# A real-life analogy: laundry day!

- It's time to do the laundry: we have 3 batches
  - **Dark**, **Light**, **Color**
  - Three steps: **Wash**, **Dry**, **Fold**
  - Time is precious, let's do this *efficiently*!

**Dark**



**Light**



**Color**



**Wash**



**Dry**



**Fold**



# A real-life analogy: laundry day!

- It's time to do the laundry: we have 3 batches
  - **Dark**, **Light**, **Color**
  - Three steps: **Wash**, **Dry**, **Fold**
  - Time is precious, let's do this *efficiently*!



**MidJourney:** 1. three baskets full of laundry, one black, one white, one colored. isometric, white background. comic style  
 2. vector drawing of different types of washing and drying machines. 3. hands folding clothes, comic style on white background.

# A real-life analogy: laundry day!

- It's time to do the laundry: we have 3 batches
  - **Dark**, **Light**, **Color**
  - Three steps: **Wash**, **Dry**, **Fold**
  - Time is precious, let's do this *efficiently*!



Wash	Dry	Fold
Dark		
Light	Dark	
Color	Light	Dark
	Color	Light

Time ↓

# A real-life analogy: laundry day!

- It's time to do the laundry: we have 3 batches
  - **Dark**, **Light**, **Color**
  - Three steps: **Wash**, **Dry**, **Fold**
  - Time is precious, let's do this *efficiently*!



Wash	Dry	Fold
Dark		
Light	Dark	
Color	Light	Dark
	Color	Light

Time ↓

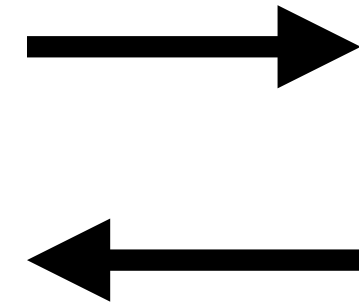
**Naïve** schedule: 9 cycles  
**Better** schedule: 4 cycles

# Latency and Throughput



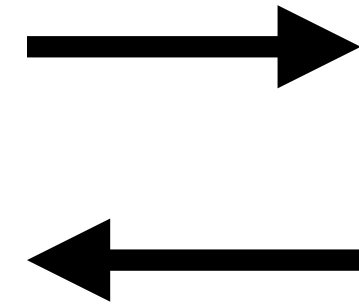
# Latency and Throughput

Latency: 1, Throughput: 1

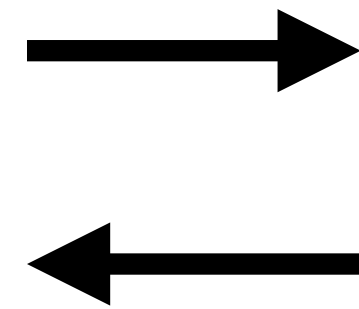


# Latency and Throughput

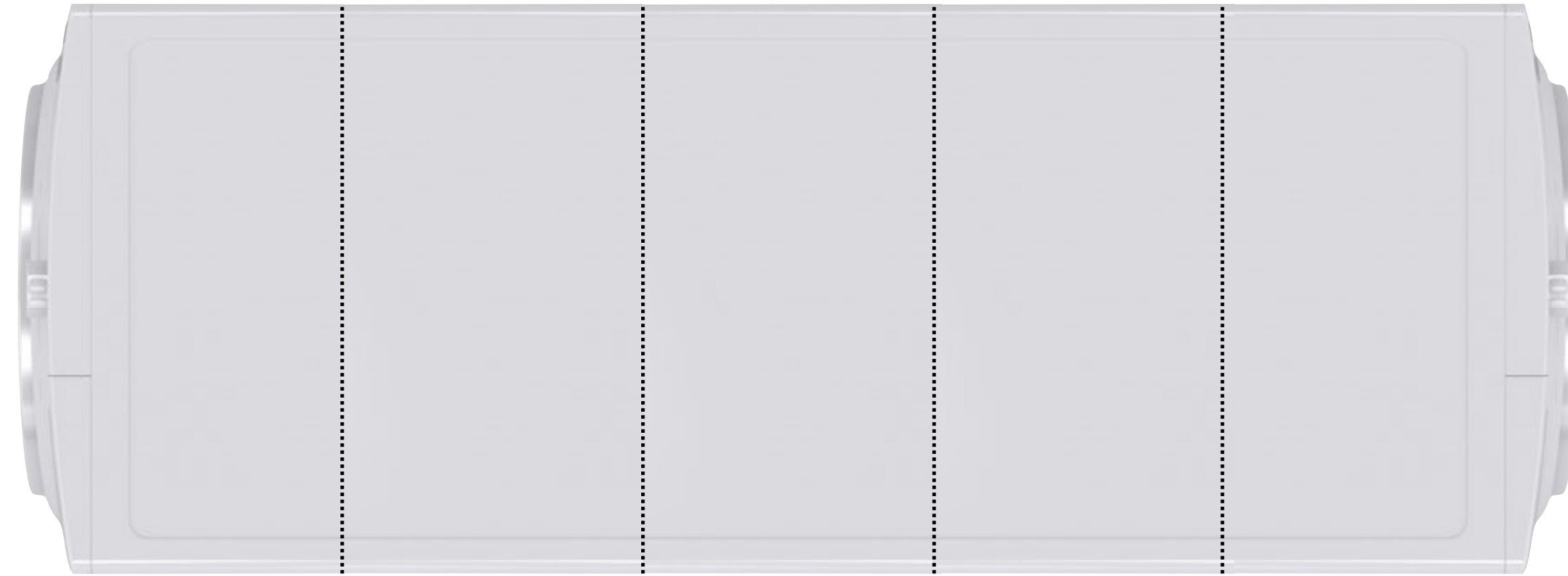
Latency: 1, Throughput: 1



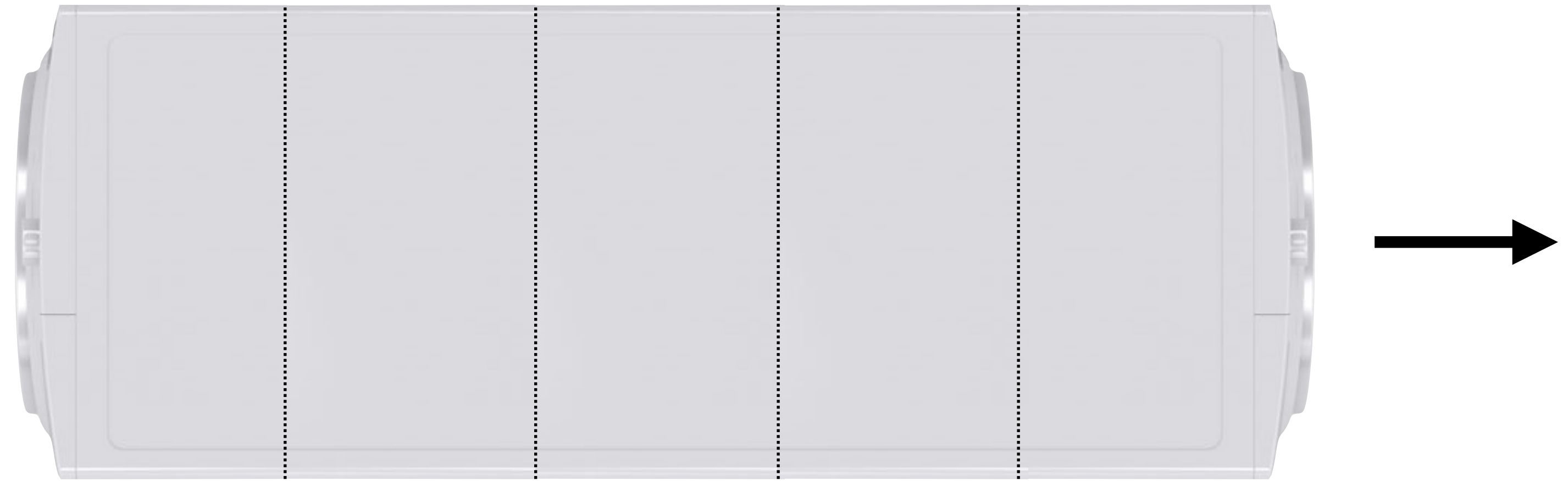
Latency: 4, Throughput: 1/4



# Pipelining *within* processing units

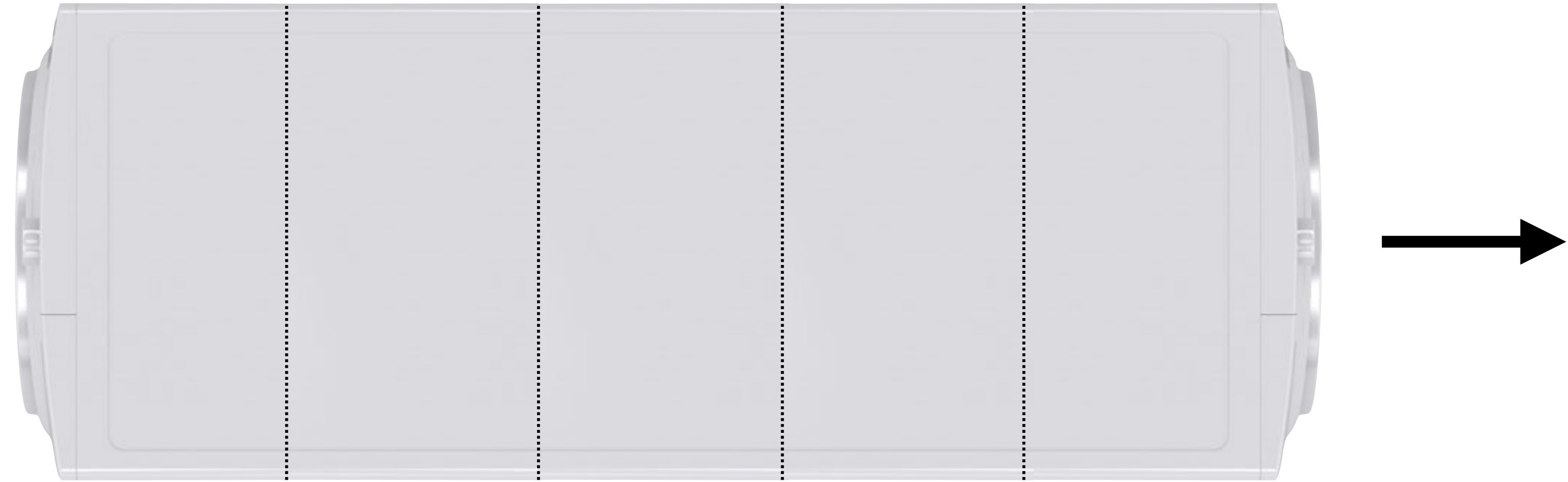


# Pipelining *within* processing units

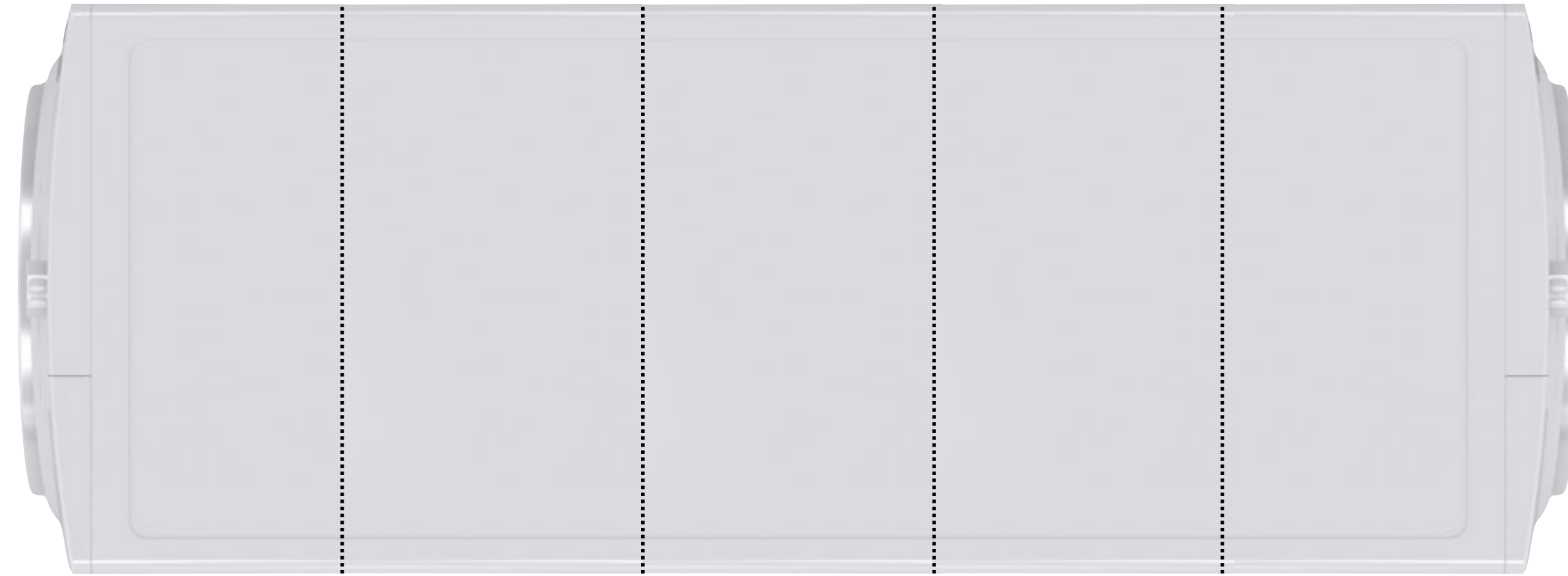


# Pipelining *within* processing units

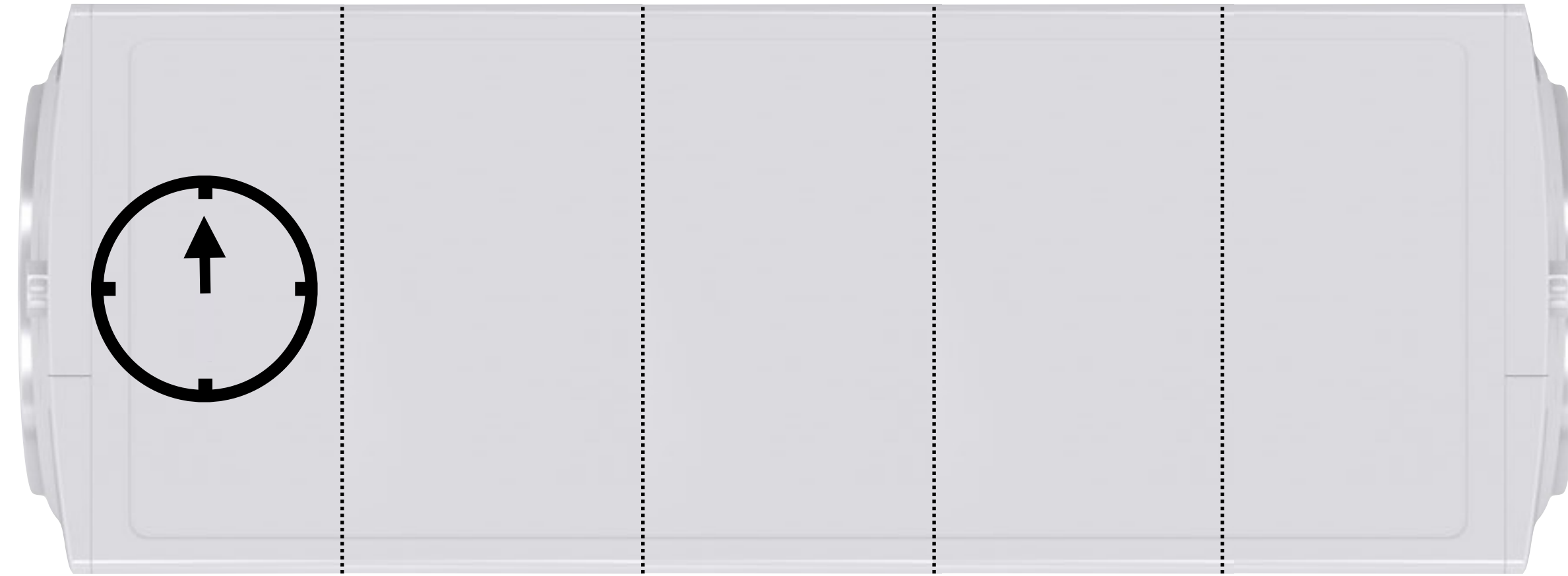
Latency: 5, Throughput: 1



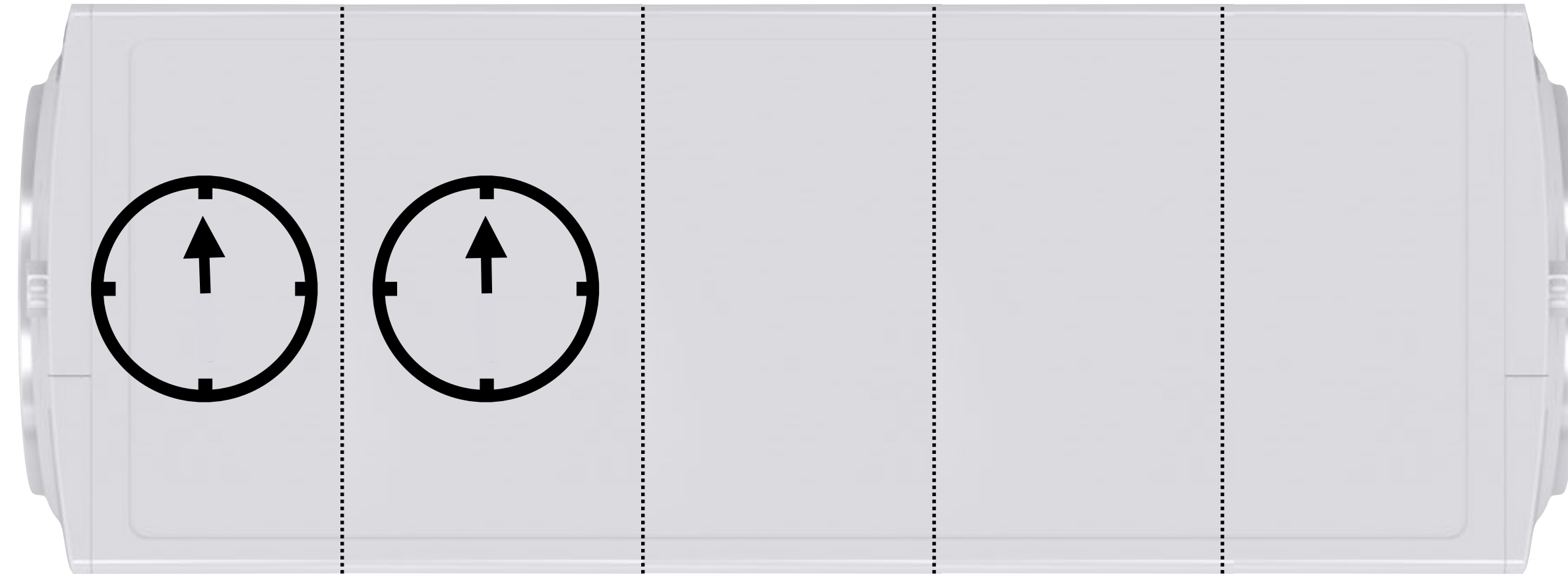
# Pipelining *within* processing units



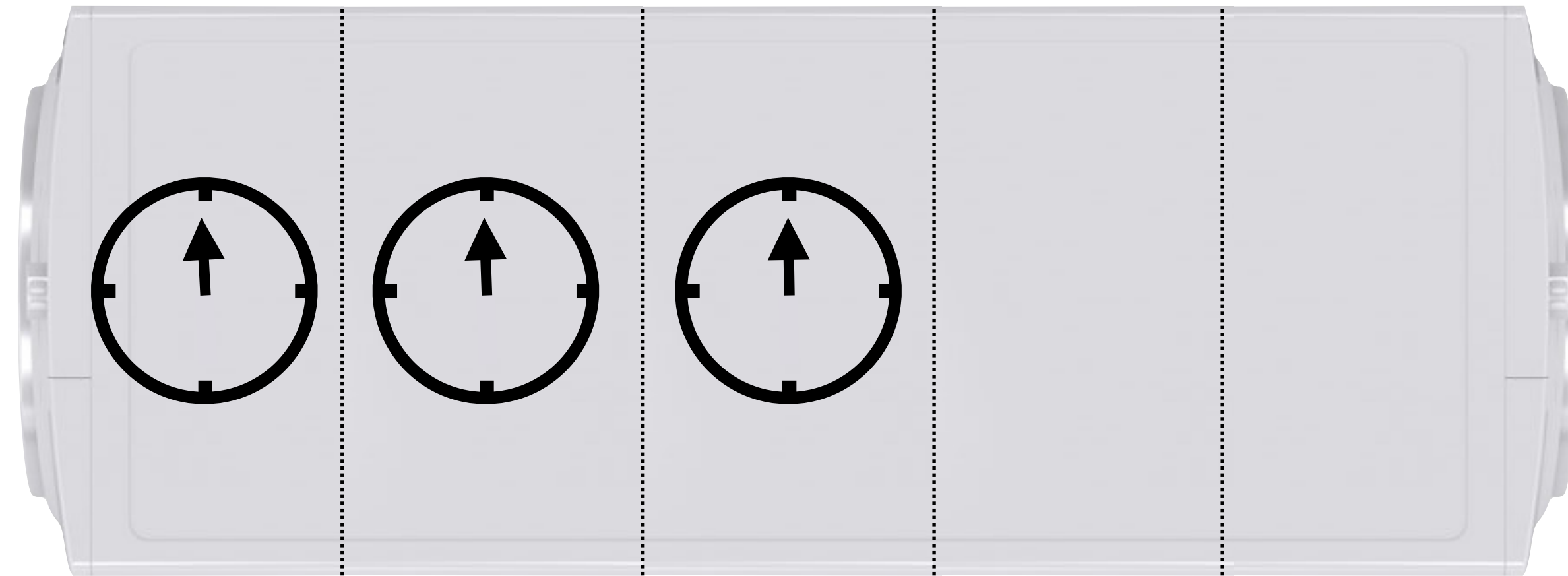
# Pipelining *within* processing units



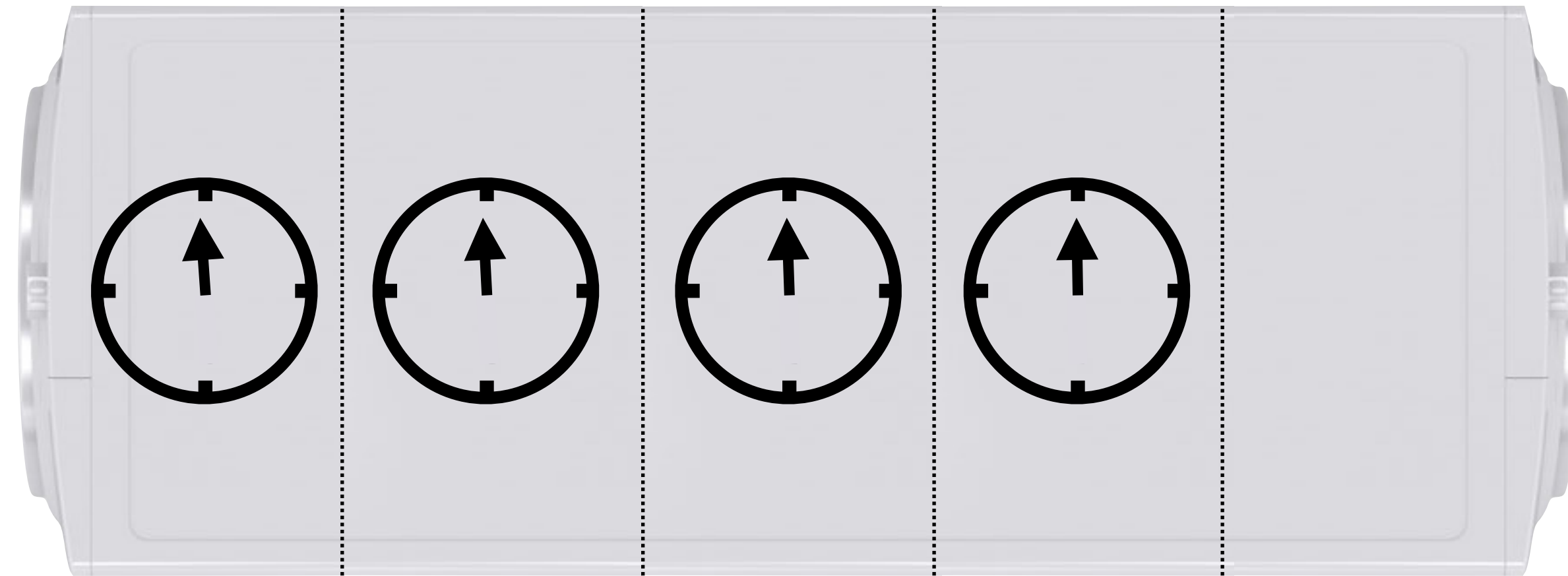
# Pipelining *within* processing units



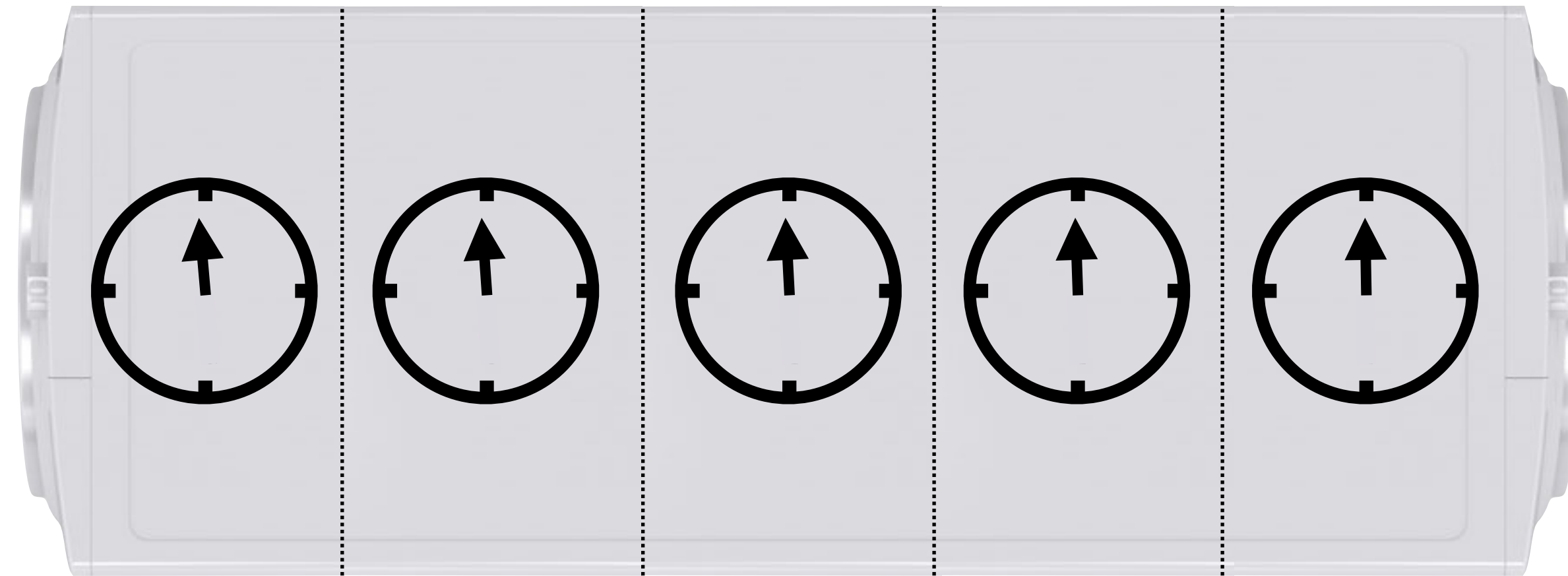
# Pipelining *within* processing units



# Pipelining *within* processing units

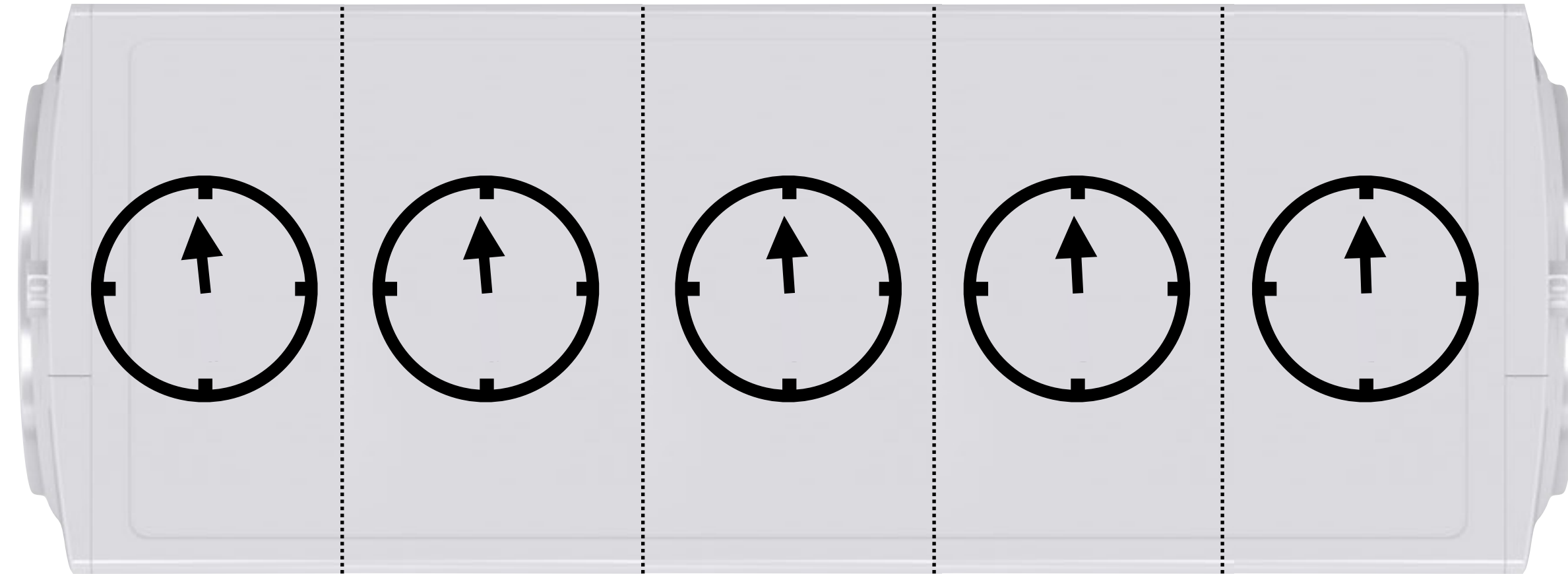


# Pipelining *within* processing units



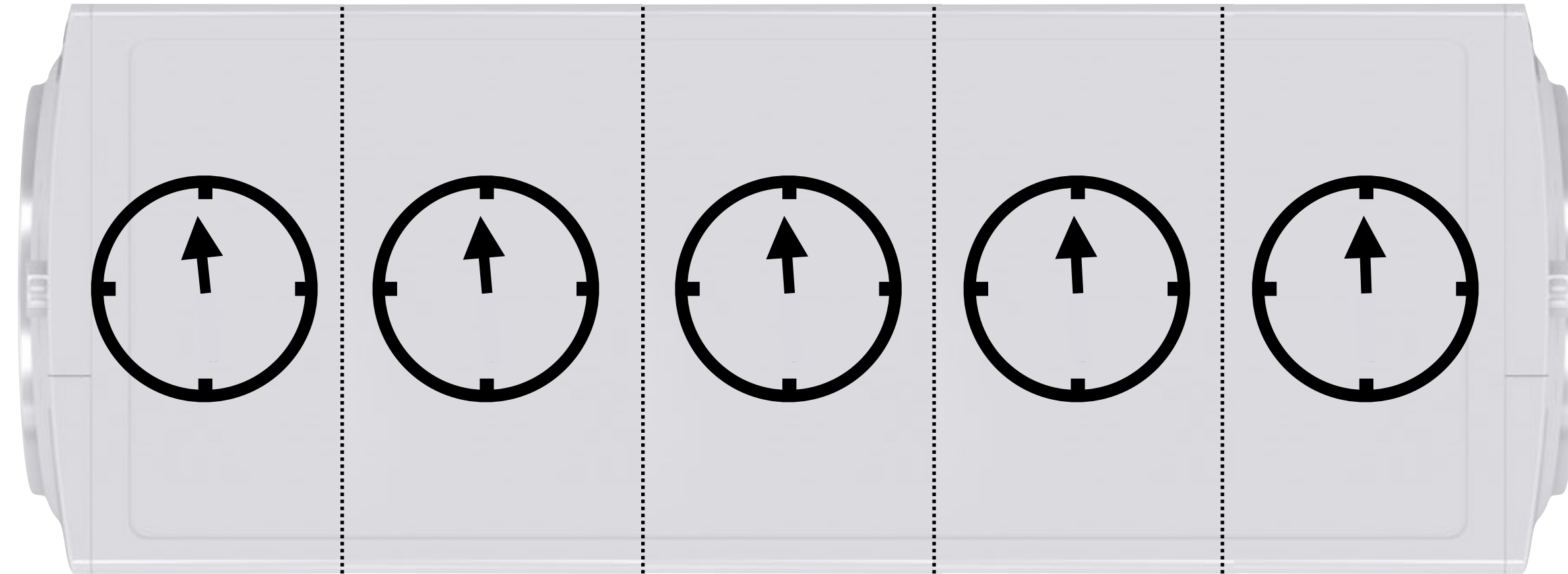
# Pipelining *within* processing units

Latency: 5, Throughput: 1

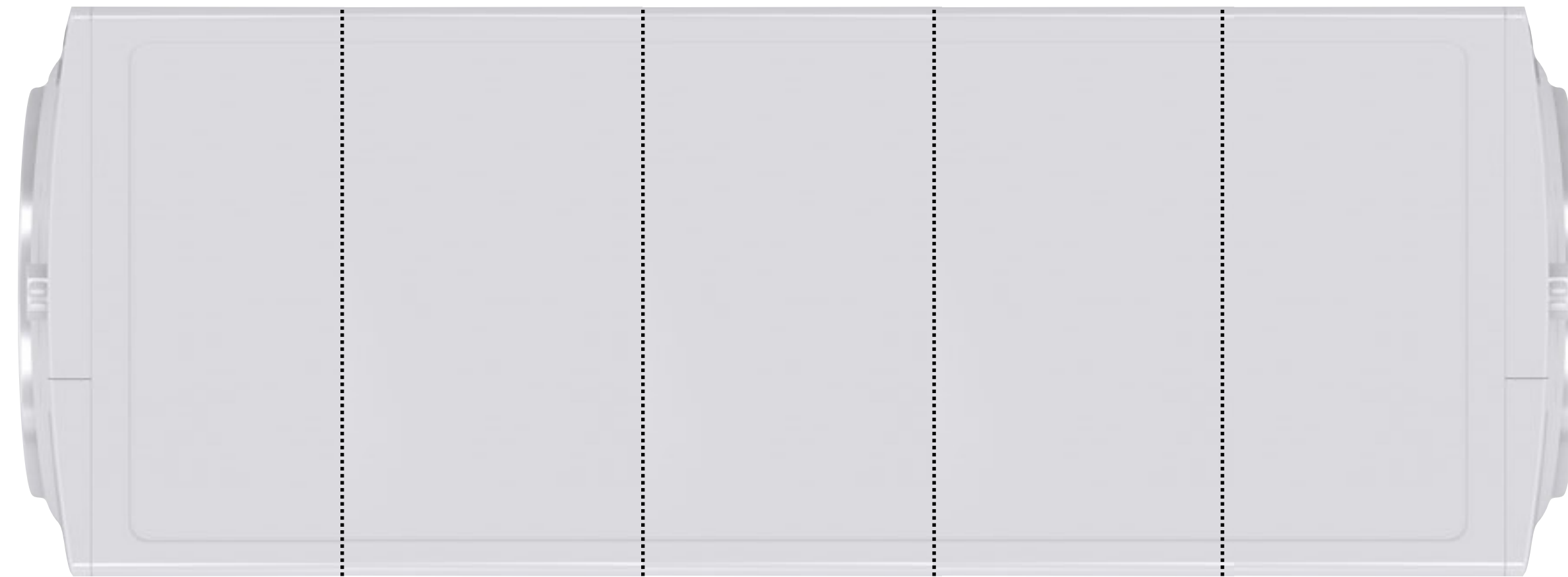


# Pipelining *within* processing units

Latency: 5, Throughput: 1



Latency: 5, Throughput: 5



# Concrete examples for a real processor

## Example data from the Intel "Ice Lake" architecture

- (Clock) **Cycle**: on a 3 GHz CPU:
    - 1 cycle = 0.33 nanoseconds.  
*Light moves 10 cm in that time.*
  - **Latency**:
    - 1 means: start an addition, result immediately available on the next cycle.
    - >1 means: need to wait several cycles.
  - **Throughput**:
    - 4 means: can start 4 additions every cycle!
    - 1/14 means: busy for 14 cycles before new input can be accepted.
- Loading** memory, **division**, and **square root** are really expensive operations.

int64 integers

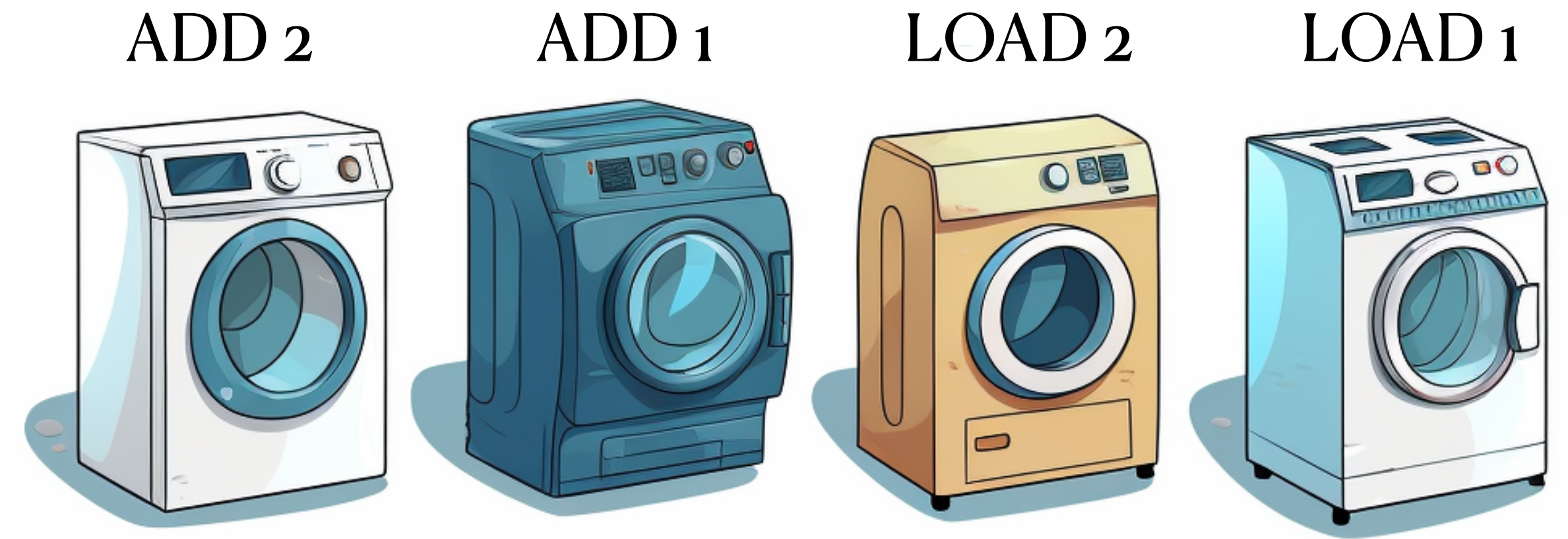
Instruction	Latency	Throughput
add, sub	1	4
mul	3	1
div	15	1/14

float32 vectors

add, sub, mul	4	2
div	17	1/10

# Tomasulo's algorithm

aka. "Superscalar execution"



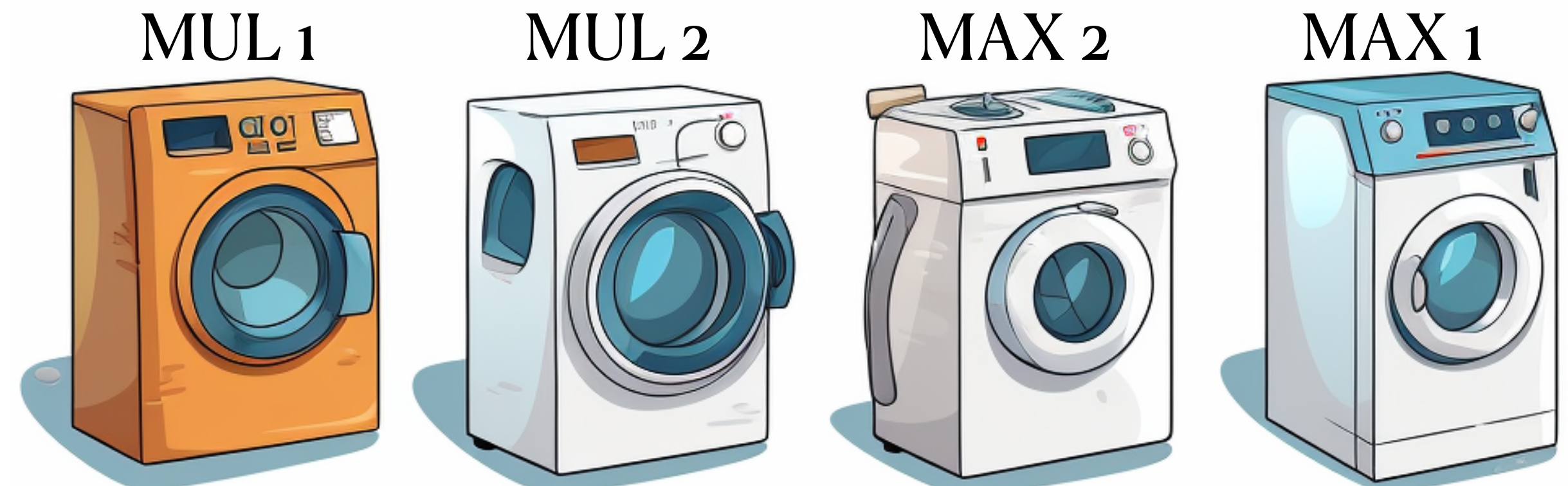
$d = d + c$



$c = \max(b, 0.0)$

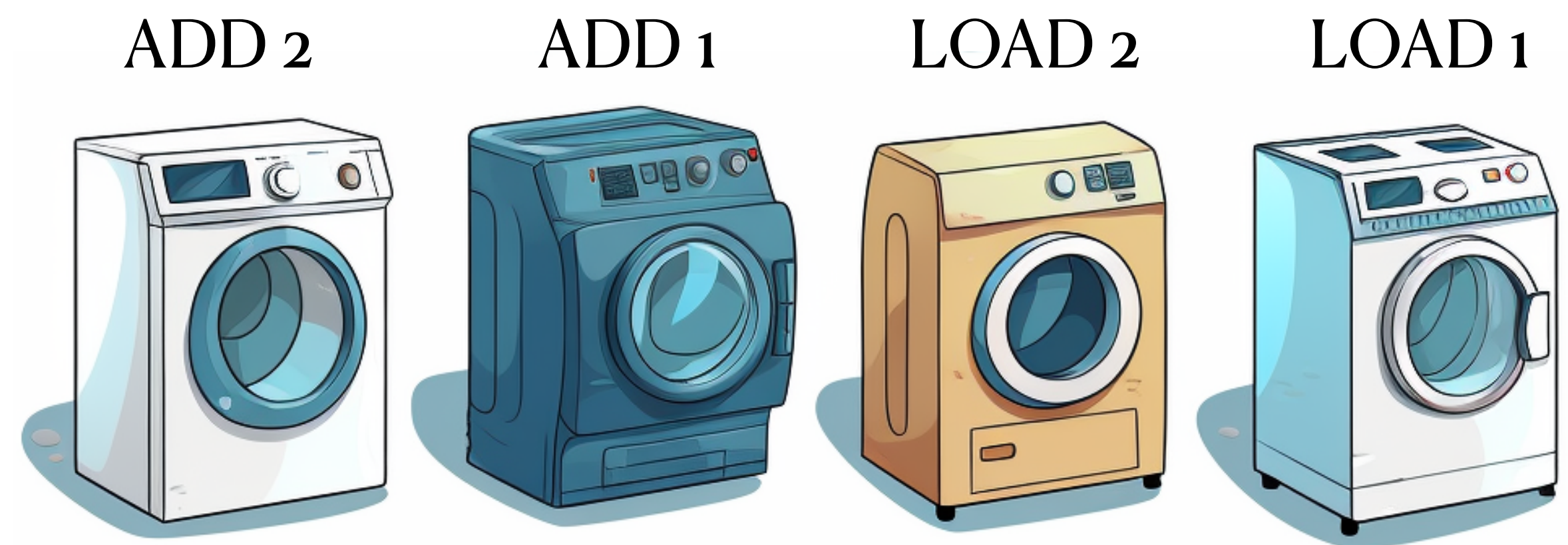


$b = a[0]$



# Tomasulo's algorithm

aka. "Superscalar execution"



Instruction stream



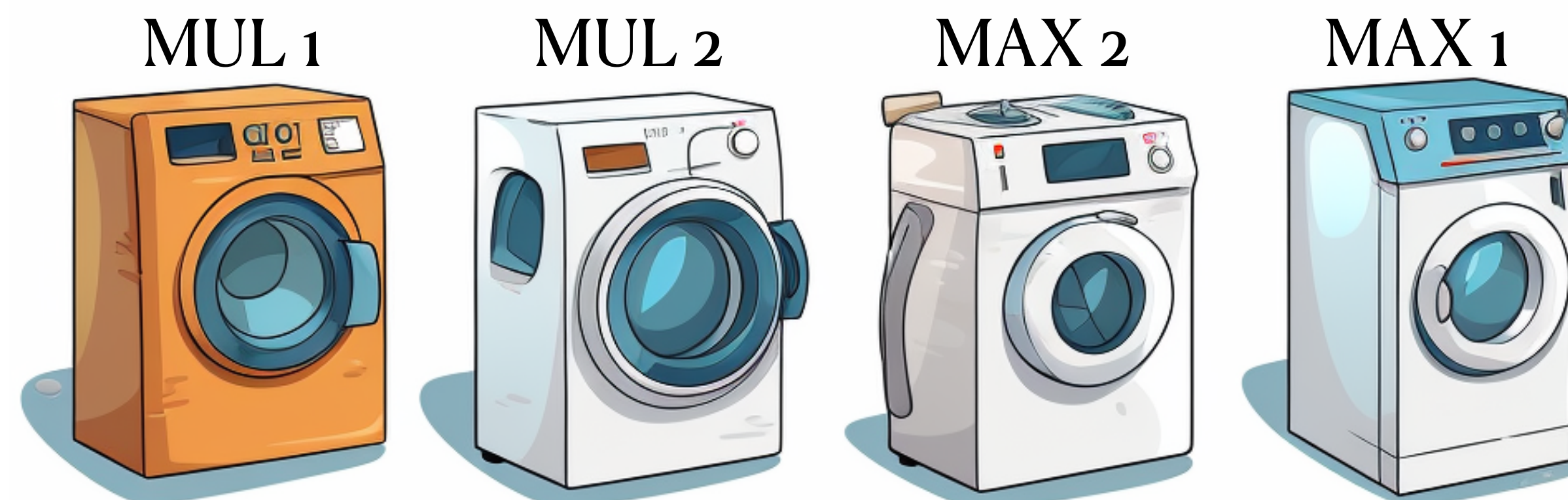
$d = d + c$



$c = \max(b, 0.0)$

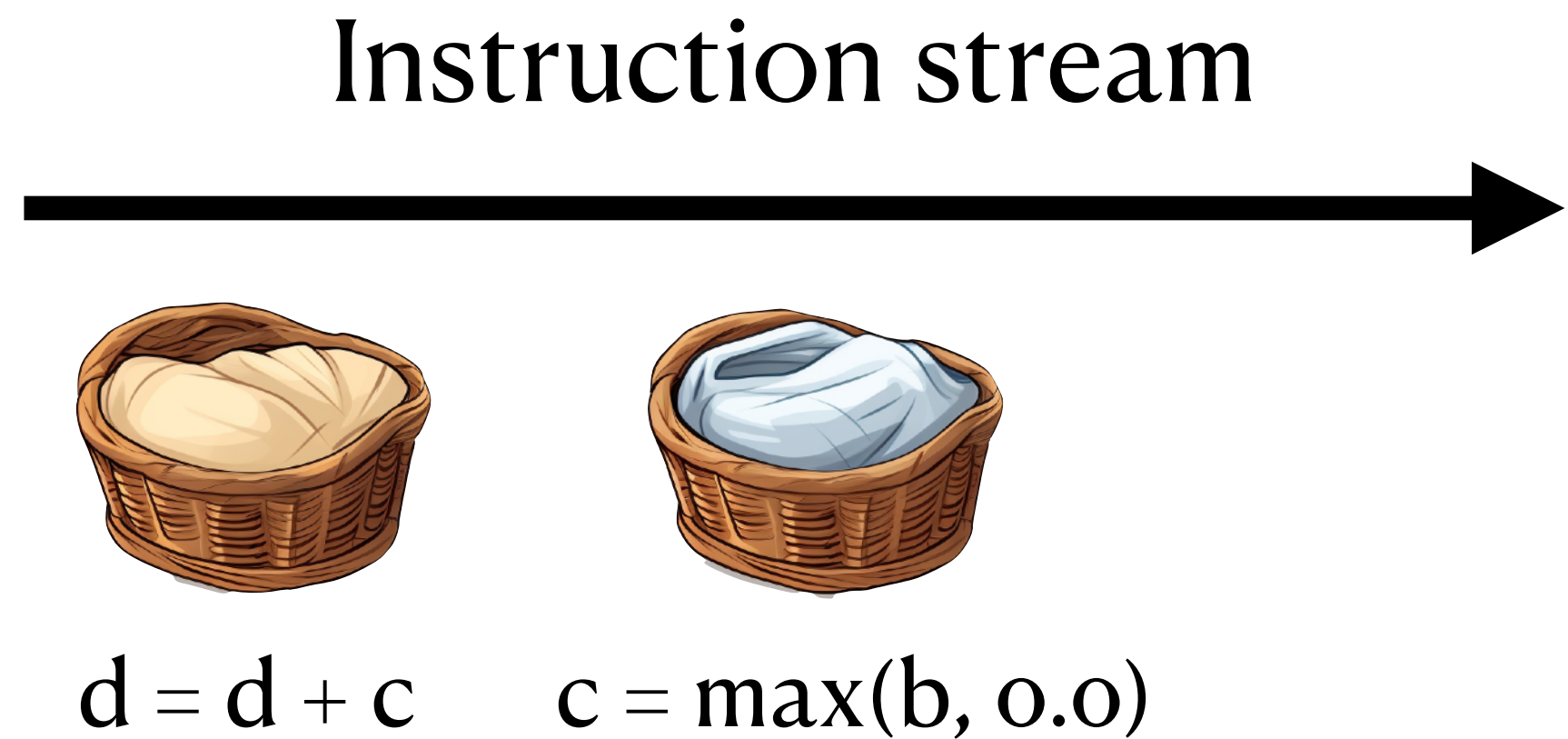


$b = a[0]$



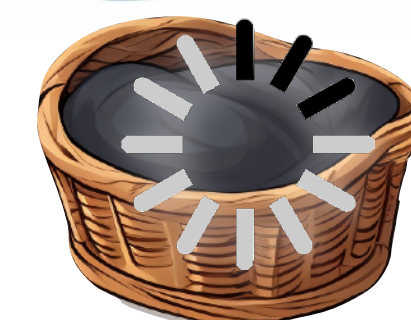
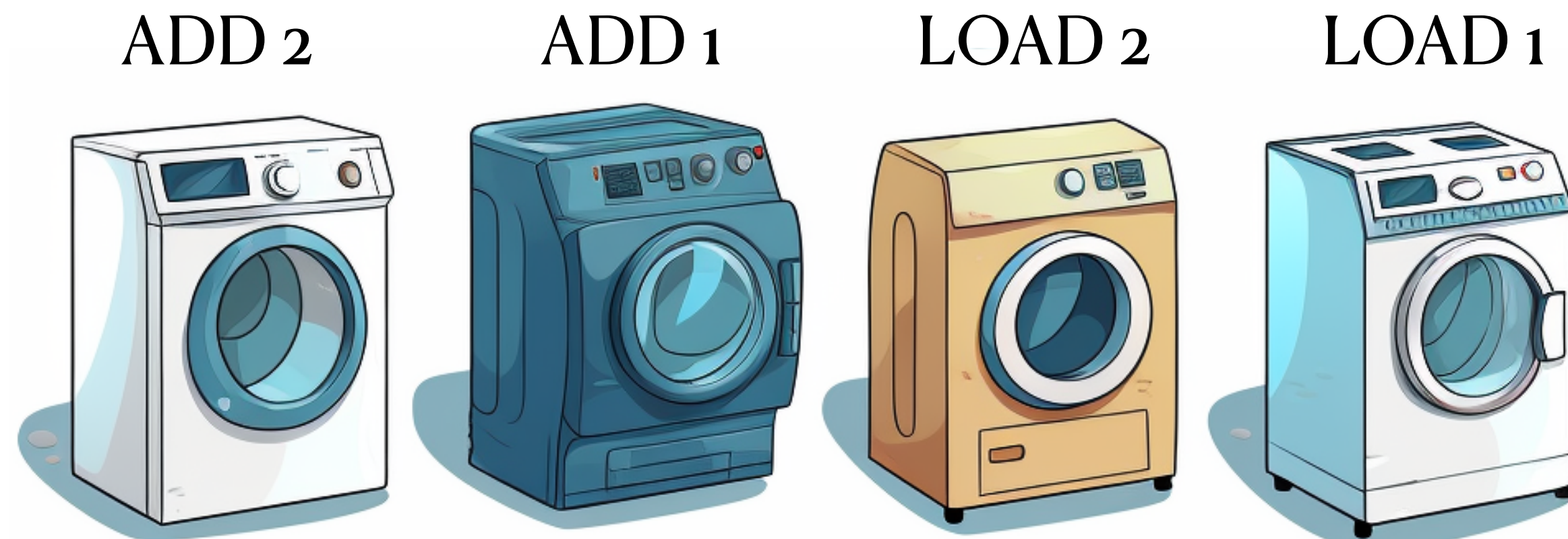
# Tomasulo's algorithm

aka. "Superscalar execution"

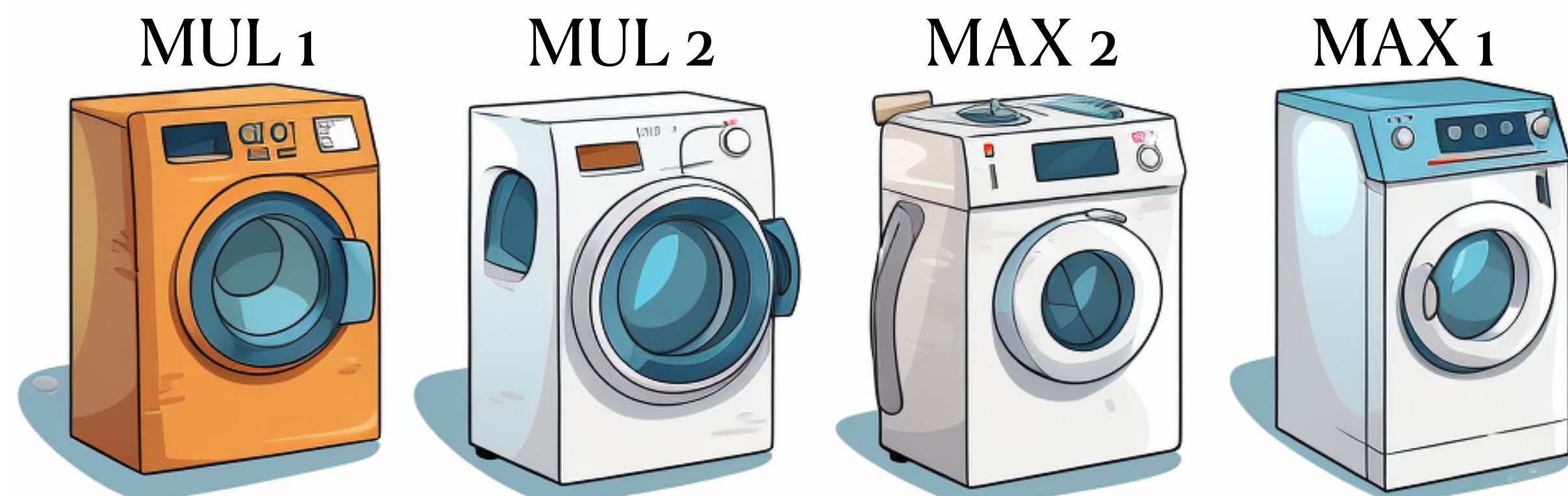


# Tomasulo's algorithm

aka. "Superscalar execution"

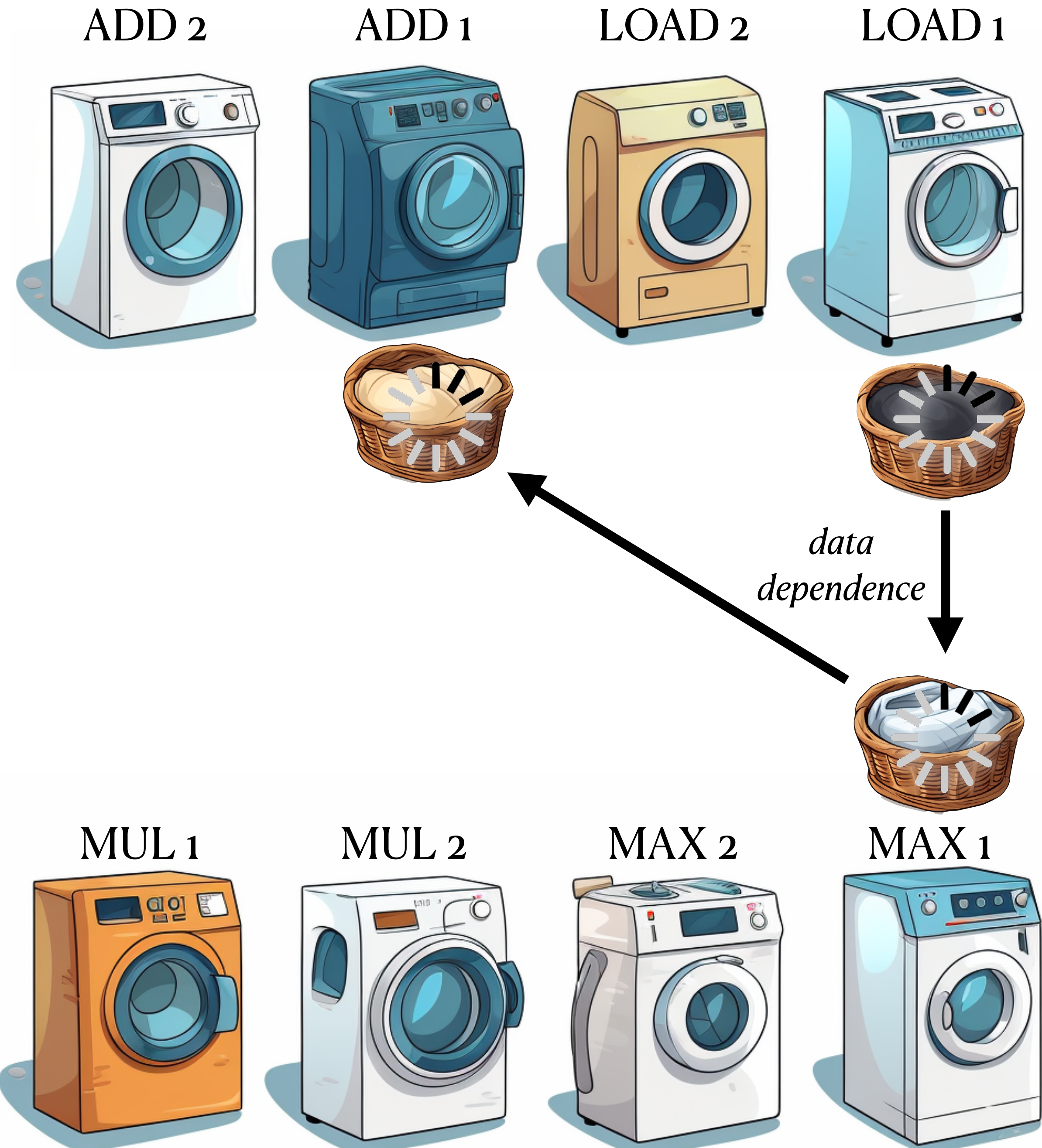


*data  
dependence*



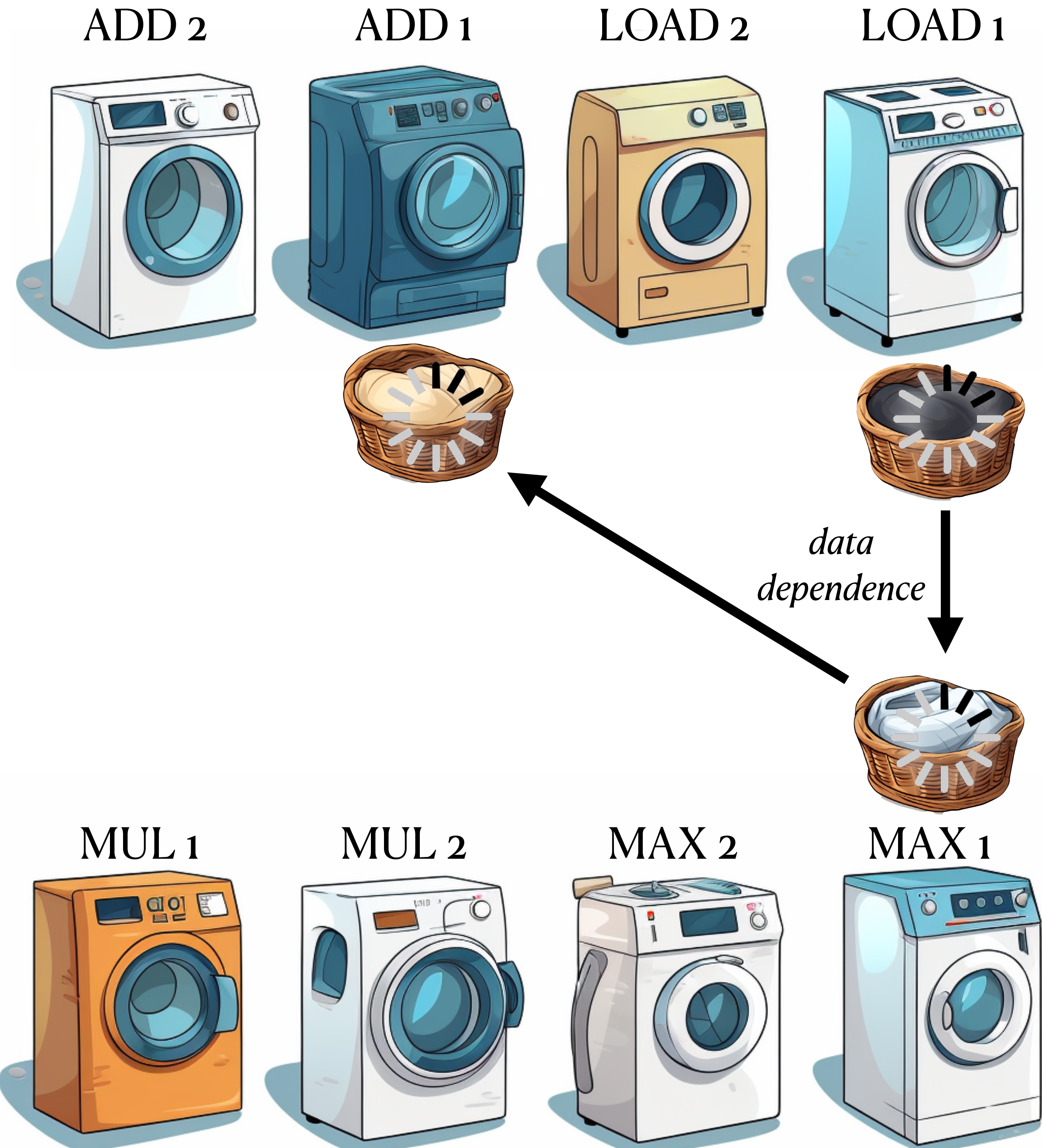
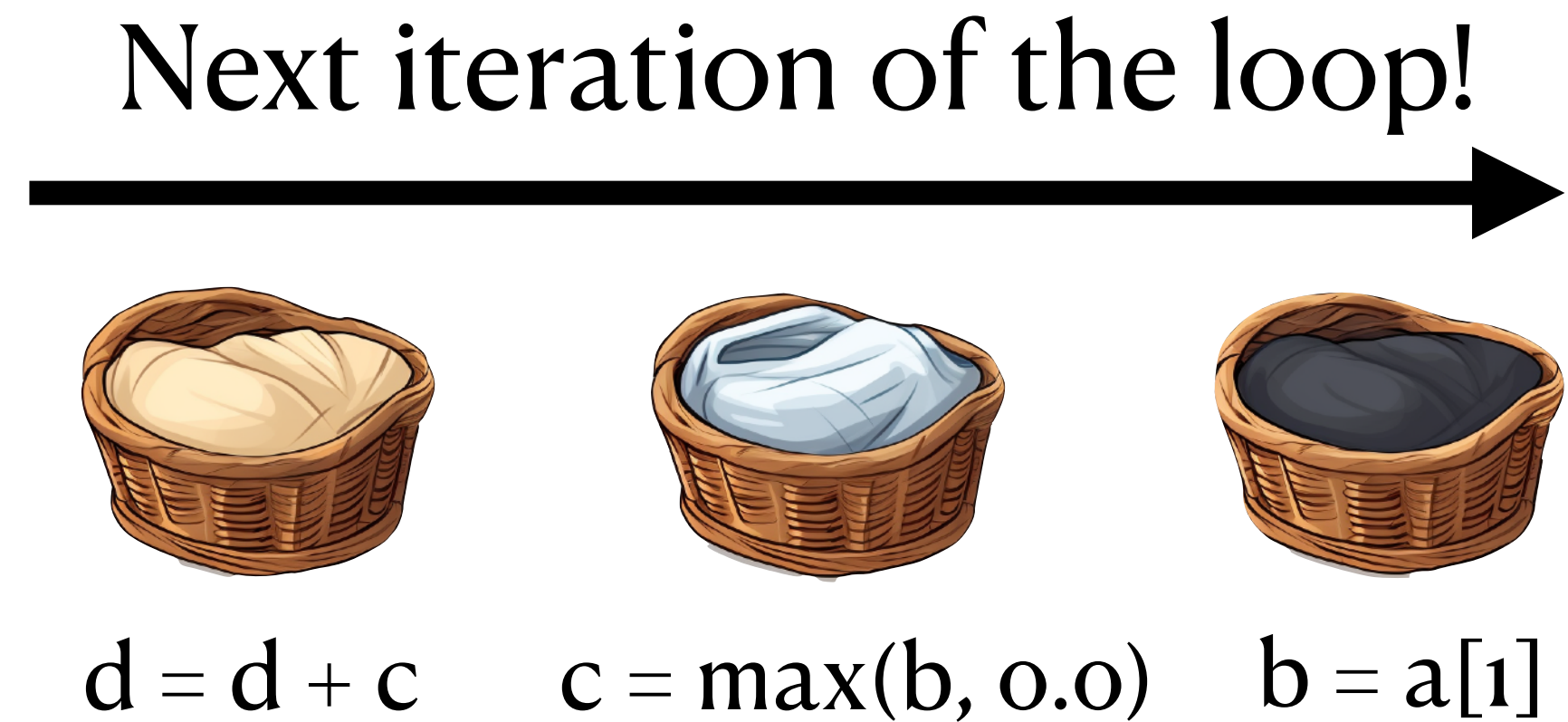
# Tomasulo's algorithm

aka. "Superscalar execution"



# Tomasulo's algorithm

aka. "Superscalar execution"



# Tomasulo's algorithm

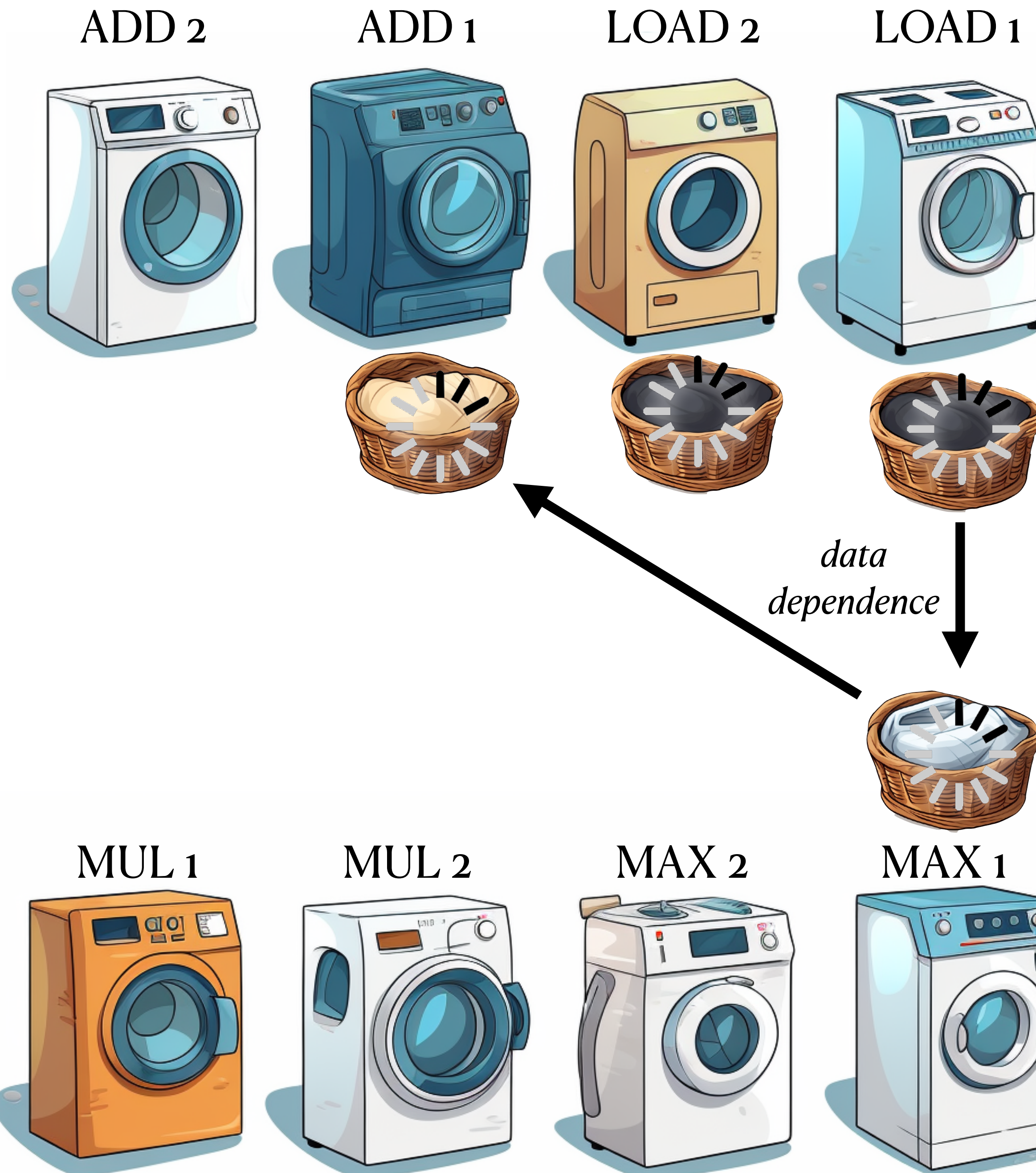
aka. "Superscalar execution"



$$d = d + c$$

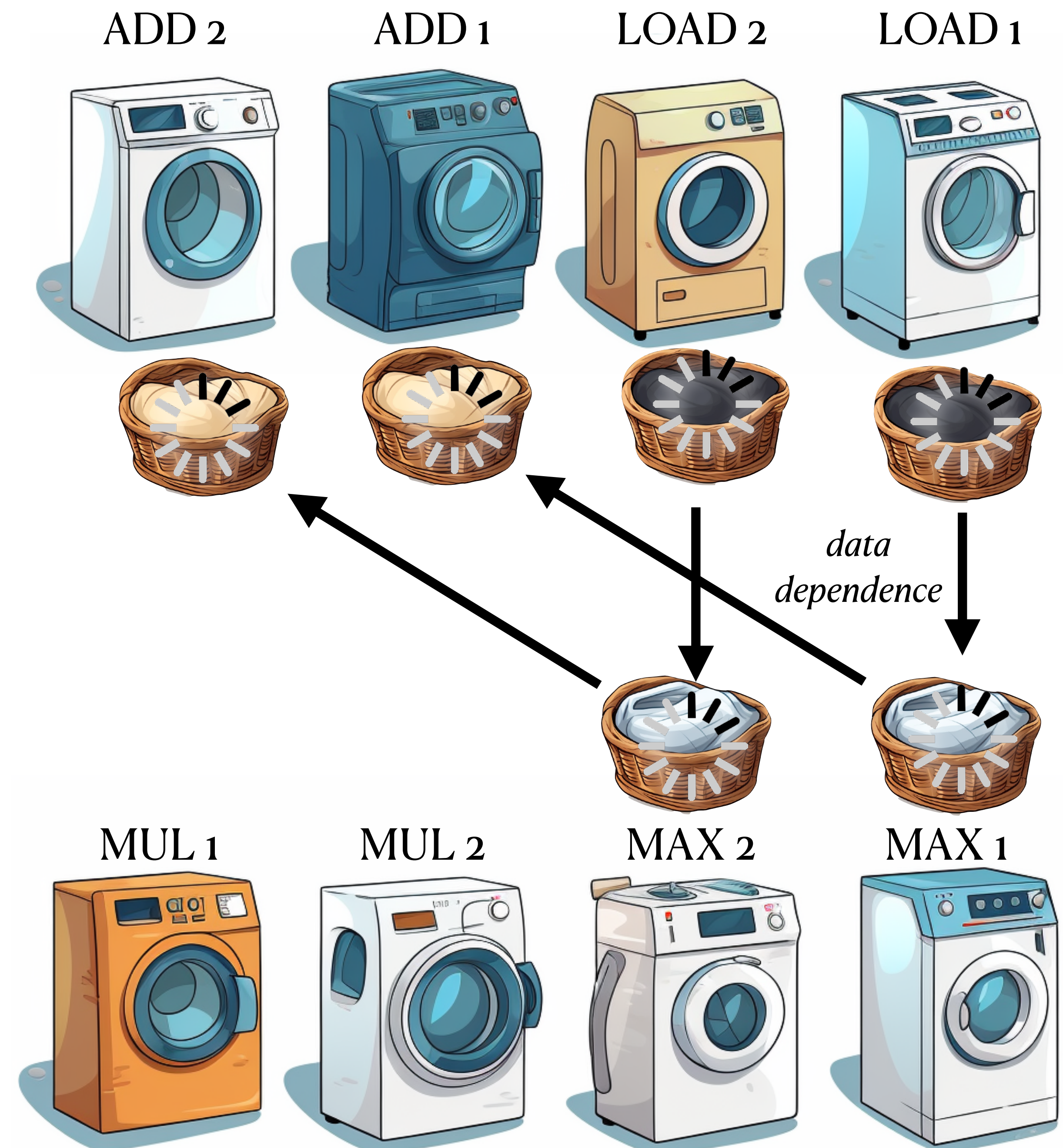


$$c = \max(b, 0.0)$$



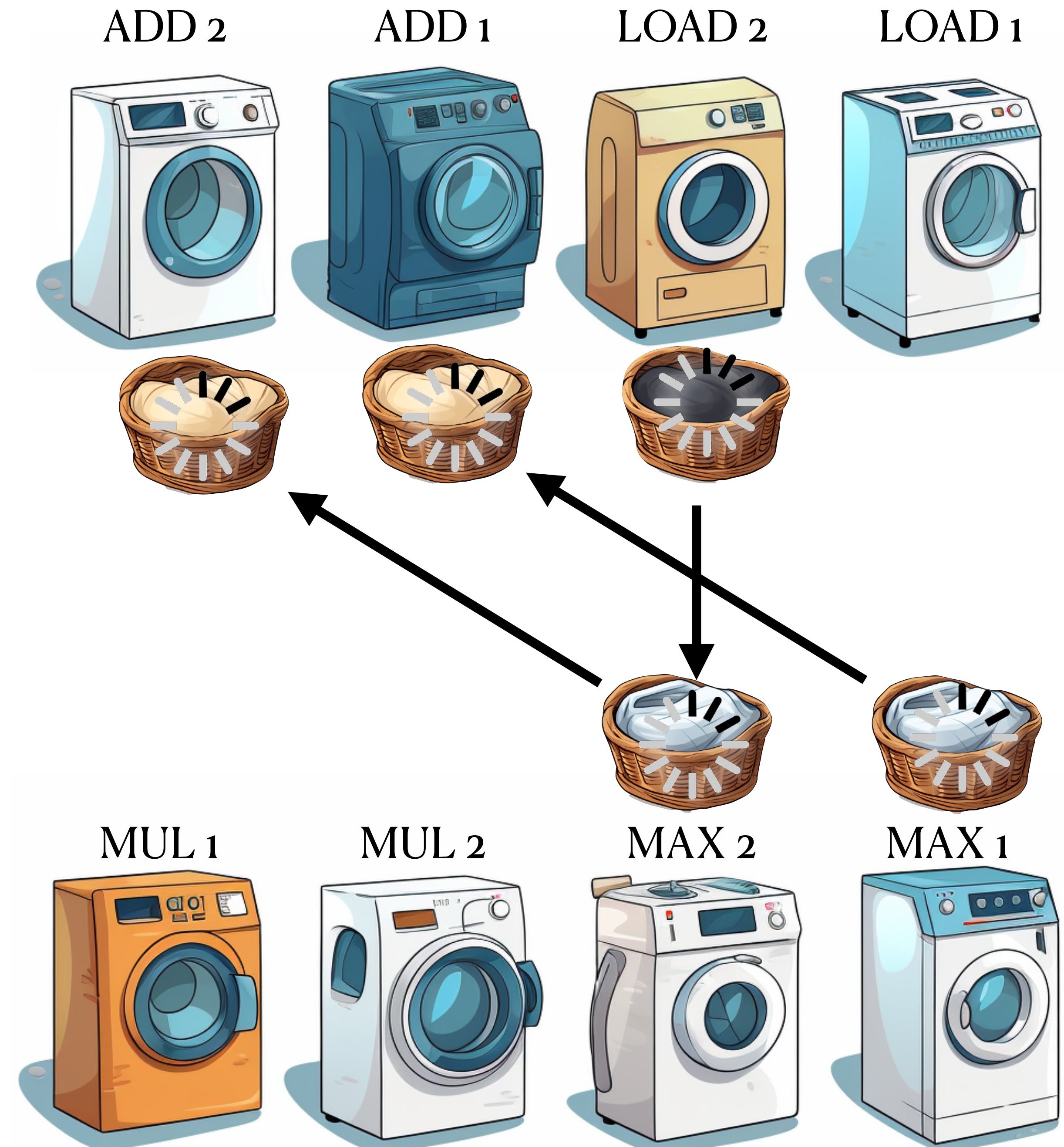
# Tomasulo's algorithm

aka. "Superscalar execution"



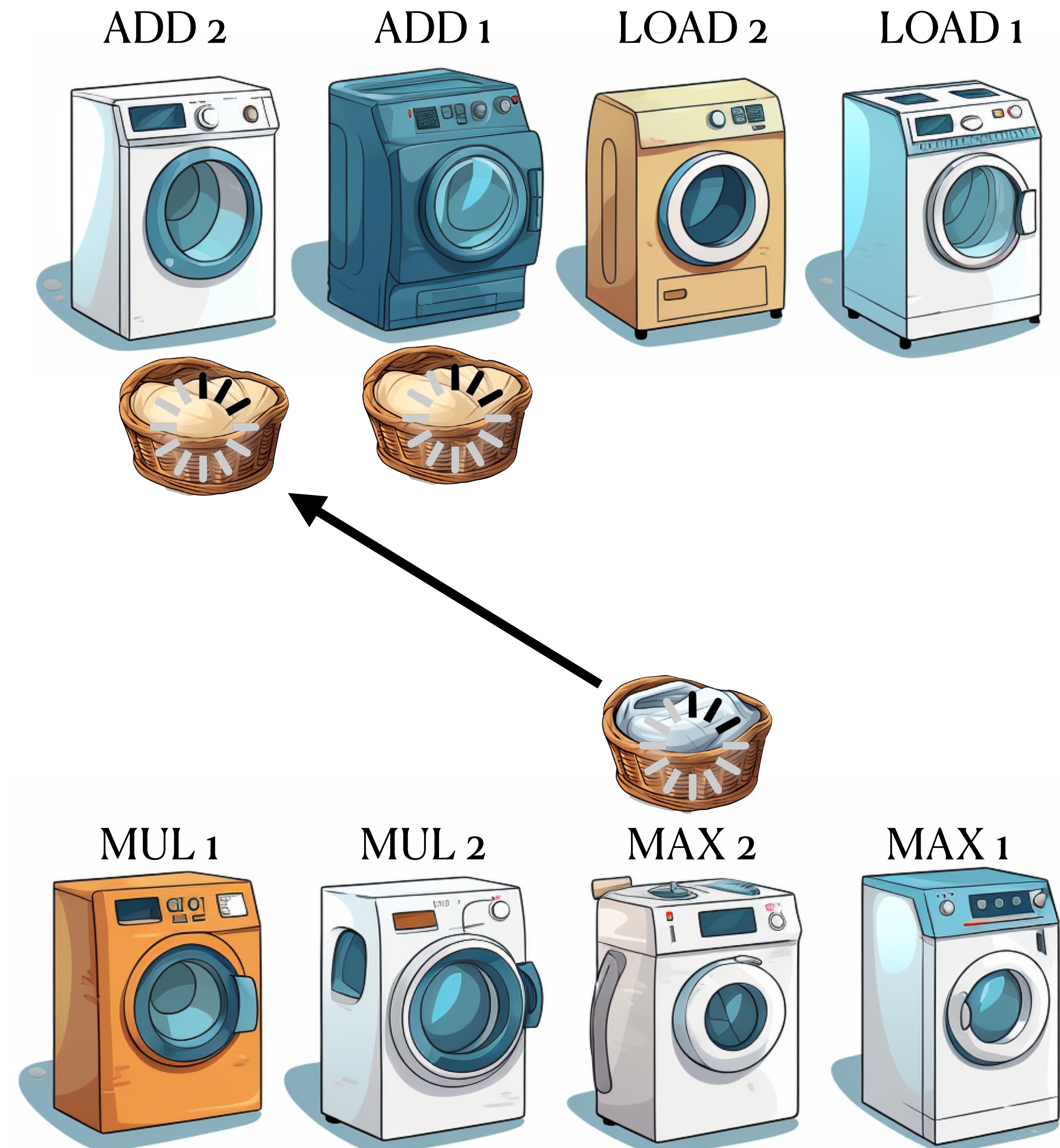
# Tomasulo's algorithm

aka. "Superscalar execution"



# Tomasulo's algorithm

aka. "Superscalar execution"



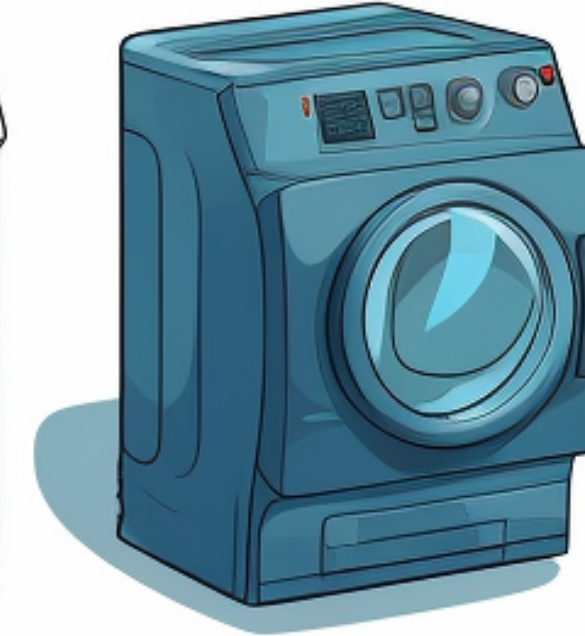
# Tomasulo's algorithm

aka. "Superscalar execution"

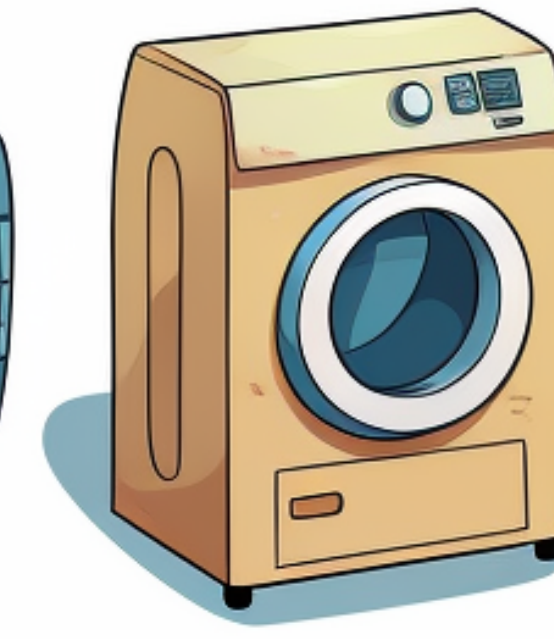
ADD 2



ADD 1



LOAD 2



LOAD 1



MUL 1



MUL 2



MAX 2



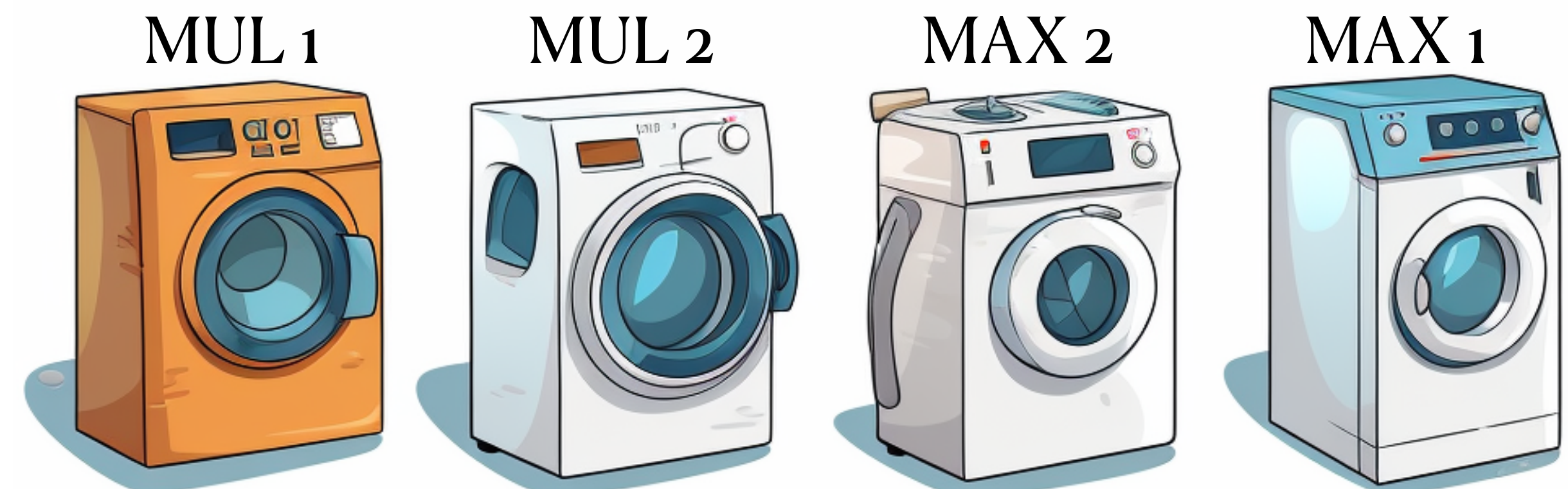
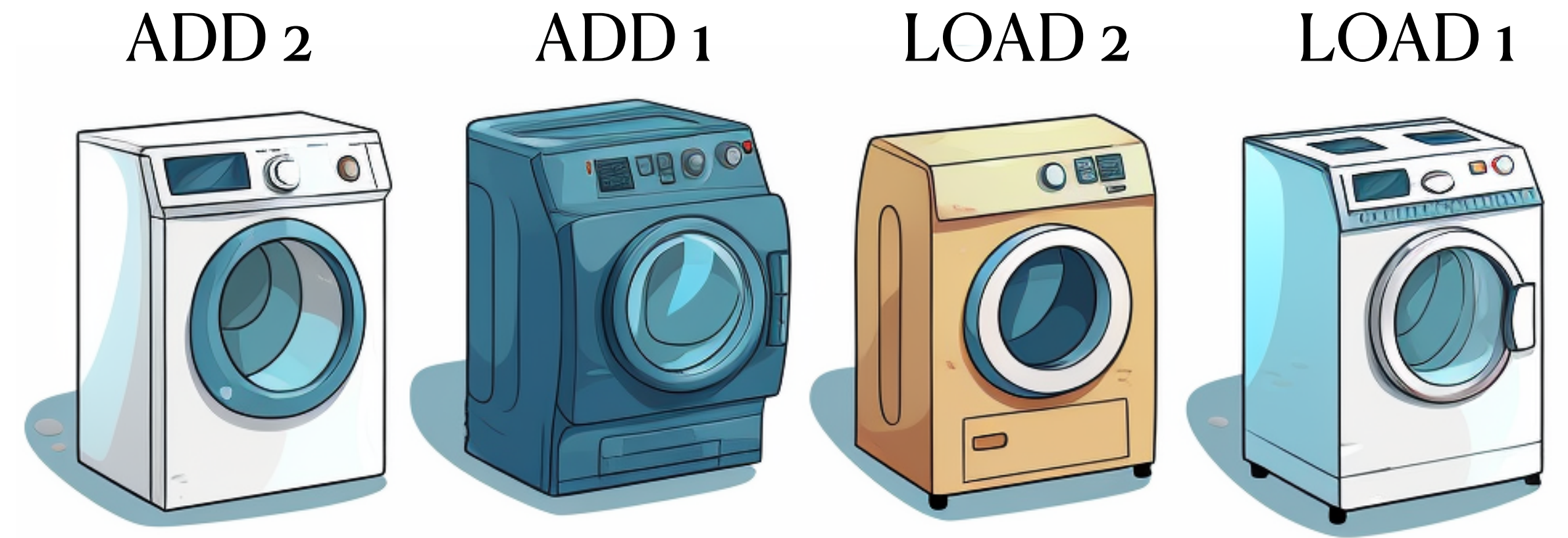
MAX 1



# Tomasulo's algorithm

aka. "Superscalar execution"

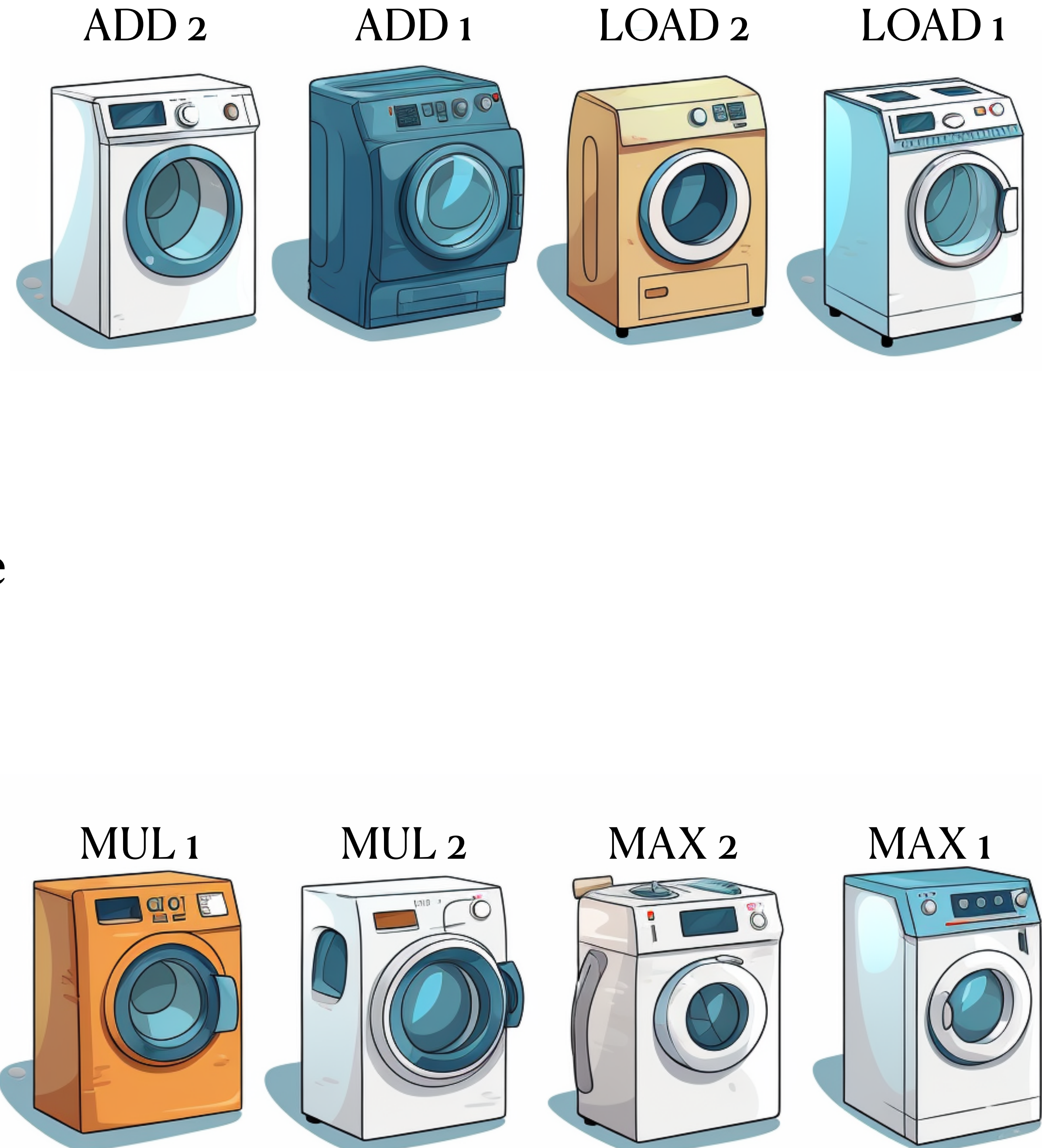
- *Aggressively parallelizes* the execution of a program. Instead of running it step after step, try to do **everything at once** while respecting data dependencies.



# Tomasulo's algorithm

aka. "Superscalar execution"

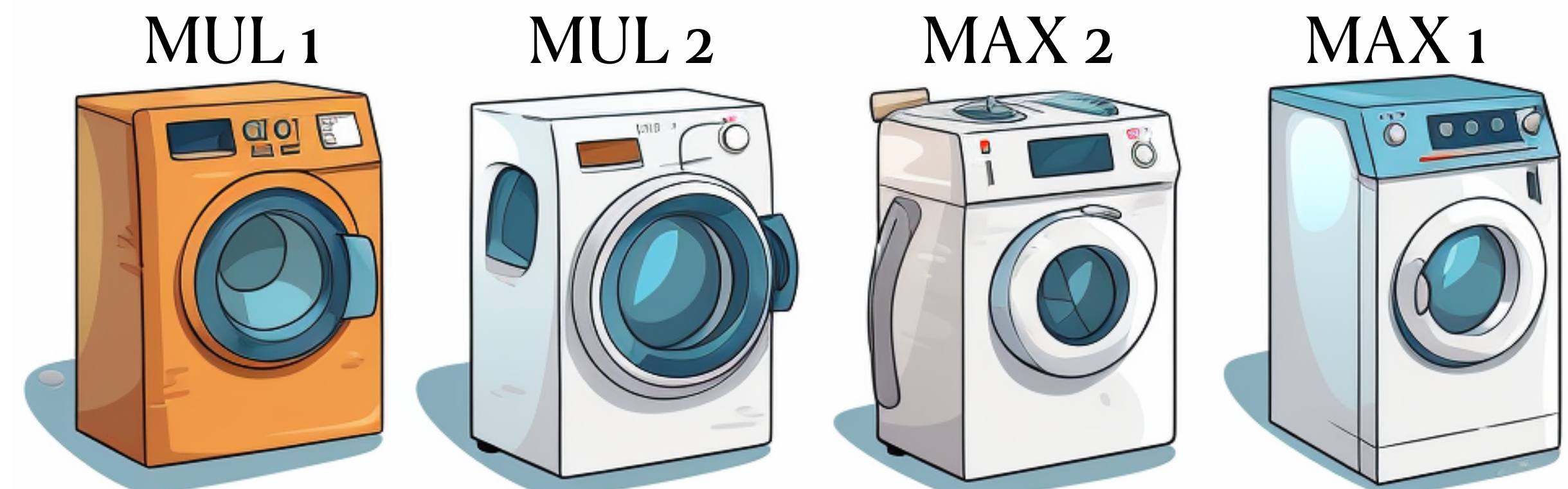
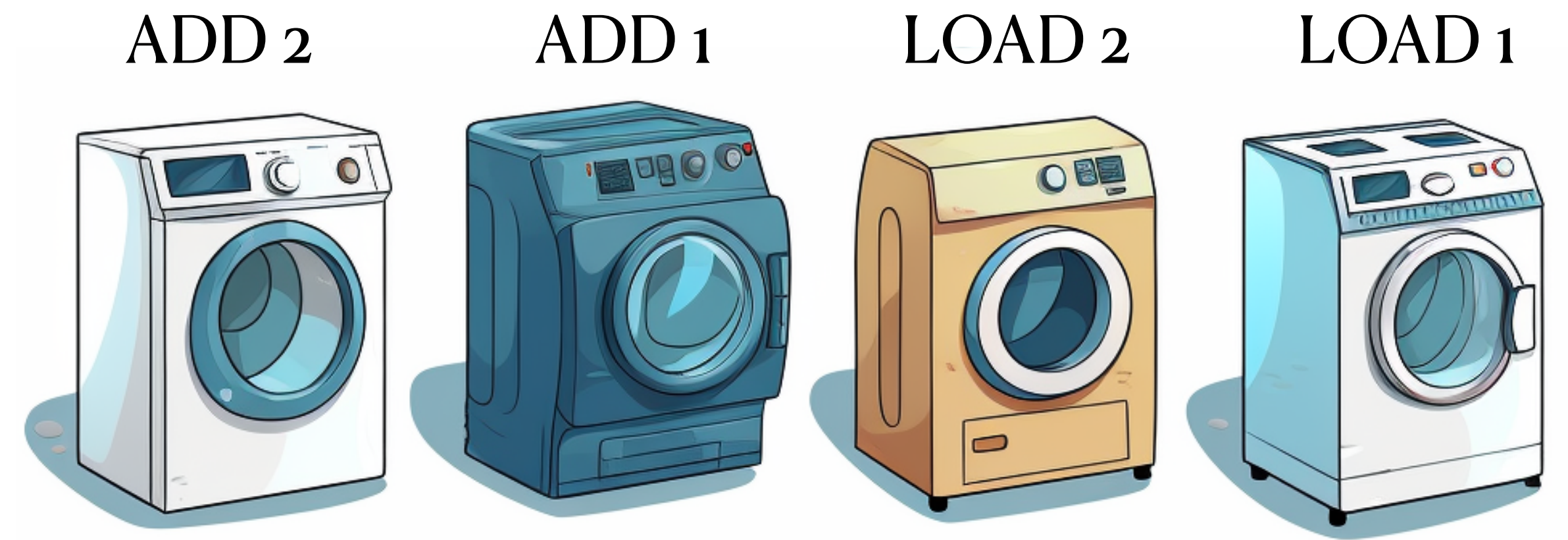
- *Aggressively parallelizes* the execution of a program. Instead of running it step after step, try to do **everything at once** while respecting data dependencies.
- Processing units must wait for inputs to be ready ("reservation stations").



# Tomasulo's algorithm

aka. "Superscalar execution"

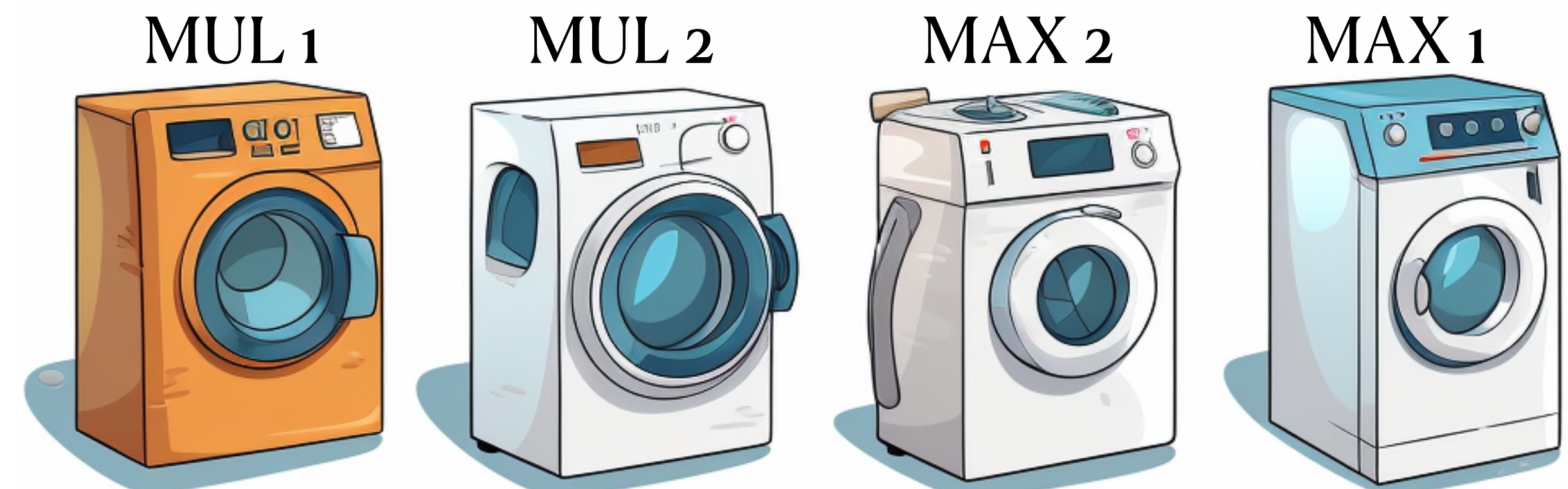
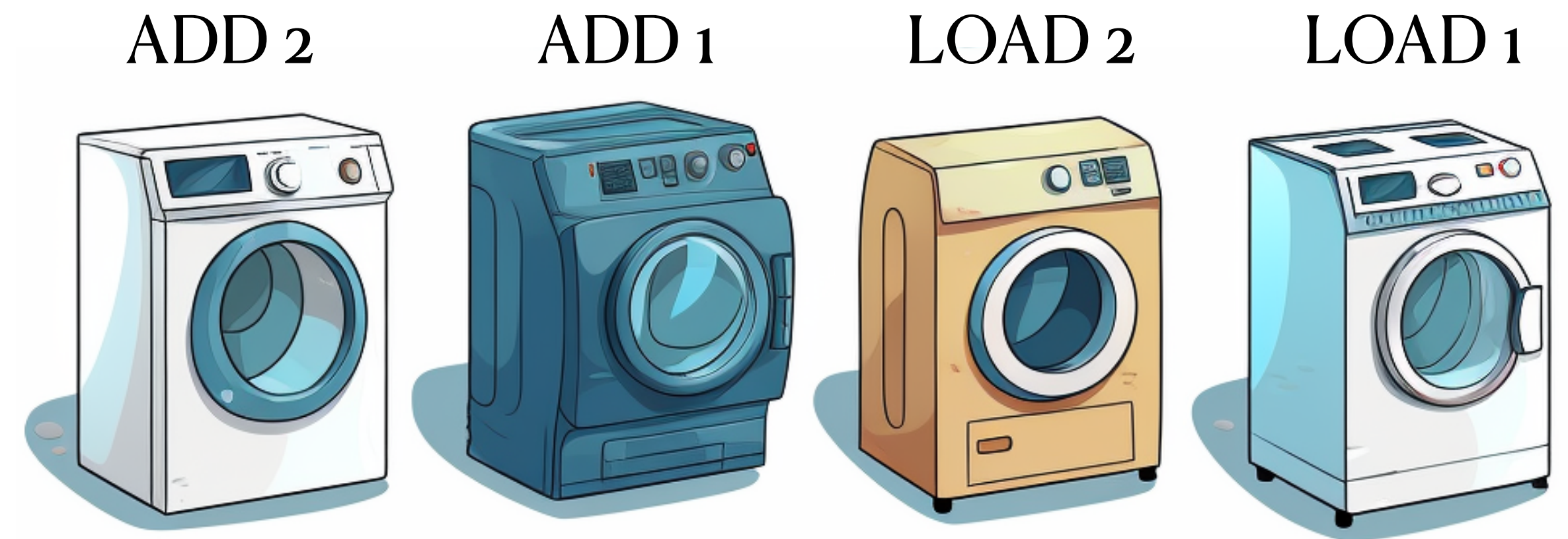
- *Aggressively parallelizes* the execution of a program. Instead of running it step after step, try to do **everything at once** while respecting data dependencies.
- Processing units must wait for inputs to be ready ("reservation stations").
- LOADs are **extremely** slow, helpful to do multiple in parallel.



# Tomasulo's algorithm

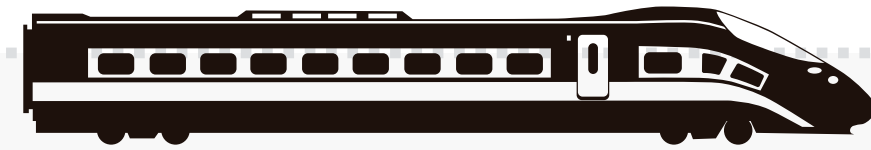
aka. "Superscalar execution"

- *Aggressively parallelizes* the execution of a program. Instead of running it step after step, try to do **everything at once** while respecting data dependencies.
- Processing units must wait for inputs to be ready ("reservation stations").
- LOADs are **extremely** slow, helpful to do multiple in parallel.
- Newer architectures (e.g. Apple M1) look at the eight (8!! 🤯) next instructions *per clock cycle*. In a loop, can run ahead several iterations 🤯🤯.



# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



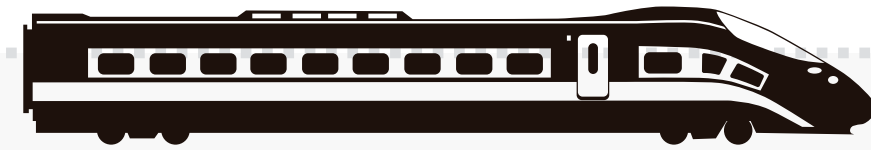
00000000	<u>C5</u>	F8	57	C0	48	85	FF	74	20	0F	1F	80	00	00	00	00
00000010	C5	FA	10	0E	C5	F2	59	0A	C5	FA	58	C1	48	83	C2	04
00000020	48	83	C6	04	48	FF	CF	75	E7	C3	.	.	.	.	.	.
00000030	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



00000000	<u>C5 F8 57 C0</u>	48	85	FF	74	20	0F	1F	80	00	00	00	00			
00000010	C5	FA	10	0E	C5	F2	59	0A	C5	FA	58	C1	48	83	C2	04
00000020	48	83	C6	04	48	FF	CF	75	E7	C3	.	.	.	.	.	.
00000030	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Executing **C5 F8 57 C0**:

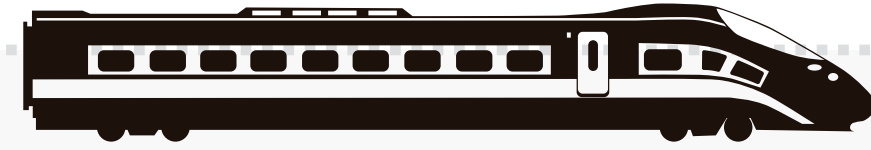
xorps xmm0, xmm0, xmm0



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



00000000	<b>C5</b>	<b>F8</b>	<b>57</b>	<b>C0</b>	48	85	FF	74	20	0F	1F	80	00	00	00	00
00000010	C5	FA	10	0E	C5	F2	59	0A	C5	FA	58	C1	48	83	C2	04
00000020	48	83	C6	04	48	FF	CF	75	E7	C3	.	.	.	.	.	.
00000030	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

**Executing C5 F8 57 C0:**

```
xorps xmm0, xmm0, xmm0
```

**Scheduling 48 85 FF 74 20:**

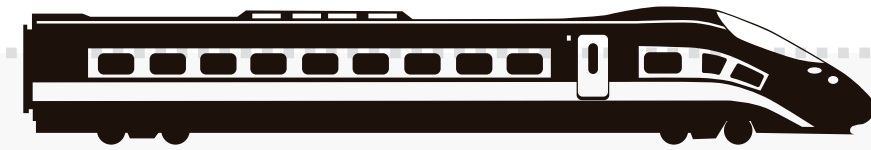
```
test rdi, rdi  
jz loop_end
```



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



00000000	<b>C5</b>	<b>F8</b>	<b>57</b>	<b>C0</b>	48	85	FF	74	20	0F	1F	80	00	00	00	00
00000010	C5	FA	10	0E	C5	F2	59	0A	C5	FA	58	C1	48	83	C2	04
00000020	48	83	C6	04	48	FF	CF	75	E7	C3	.	.	.	.	.	.
00000030	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

**Executing C5 F8 57 C0:**

```
xorps xmm0, xmm0, xmm0
```

**Decoding**

```
0F 1F 80 00 00
```

```
00 00 C5 FA 10 0E:
```

???????

???????

???????

**Scheduling 48 85 FF 74 20:**

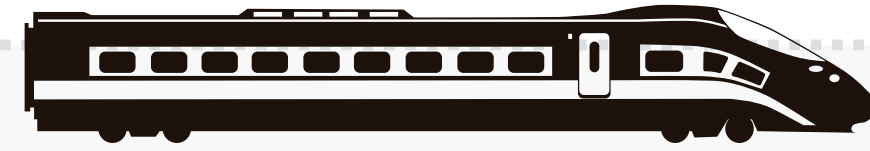
```
test rdi, rdi
jz loop_end
```



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



00000000	<b>C5</b>	<b>F8</b>	<b>57</b>	<b>C0</b>	48	85	FF	74	20	0F	1F	80	00	00	00	00
00000010	C5	FA	10	0E	C5	F2	59	0A	C5	FA	58	C1	48	83	C2	04
00000020	48	83	C6	04	48	FF	CF	75	E7	C3	.	.	.	.	.	.
00000030	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

**Executing C5 F8 57 C0:**

```
xorps xmm0, xmm0, xmm0
```

**Decoding**

```
0F 1F 80 00 00
```

```
00 00 C5 FA 10 0E:
```

???????

???????

???????

**Scheduling 48 85 FF 74 20:**

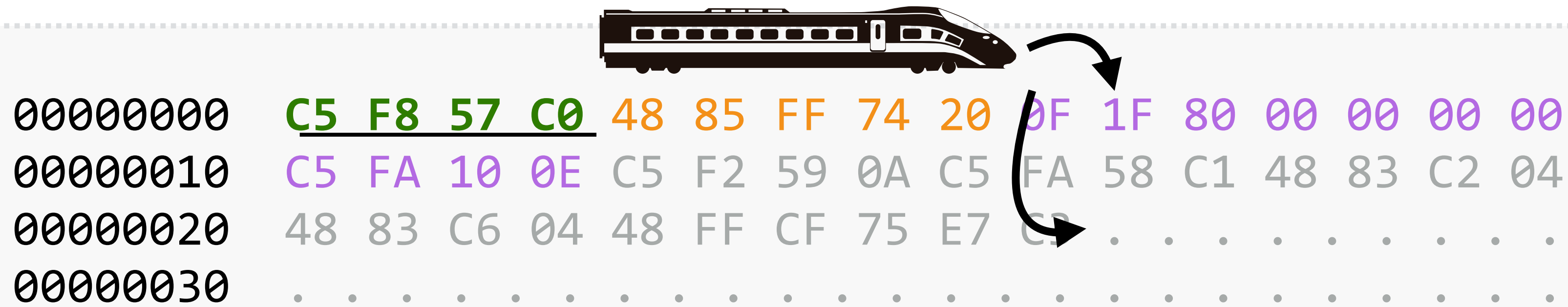
```
test rdi, rdi
jz loop_end
```



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



**Executing C5 F8 57 C0:**

```
xorps xmm0, xmm0, xmm0
```

**Decoding**

```
0F 1F 80 00 00
```

```
00 00 C5 FA 10 0E:
```

???????

???????

???????

**Scheduling 48 85 FF 74 20:**

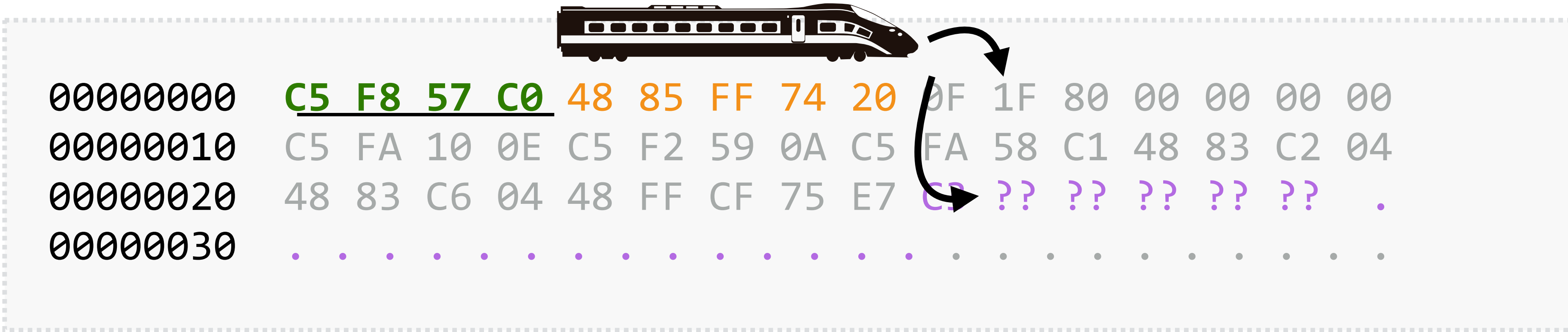
```
test rdi, rdi
jz loop_end
```



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



**Executing C5 F8 57 C0:**  
xorps xmm0, xmm0, xmm0

**Decoding** 0F 1F 80 00 00  
00 00 C5 FA 10 0E:

**Scheduling** 48 85 FF 74 20:  
test rdi, rdi  
jz loop\_end

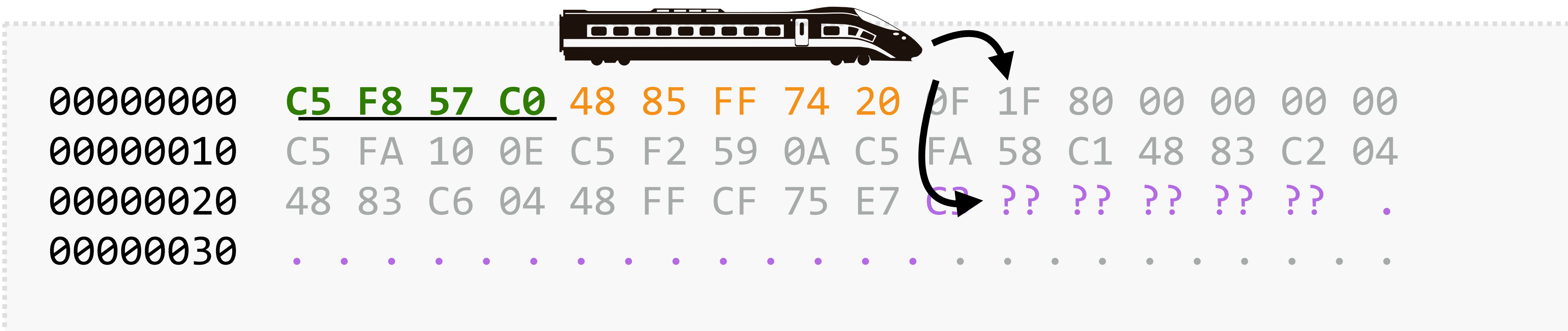
??????  
??????  
??????



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.

# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



**Executing C5 F8 57 C0:**

```
xorps xmm0, xmm0, xmm0
```

**Decoding**

```
0F 1F 80 00 00
```

```
00 00 C5 FA 10 0E:
```

???????

???????

???????

**Scheduling 48 85 FF 74 20:**

```
test rdi, rdi
jz loop_end
```

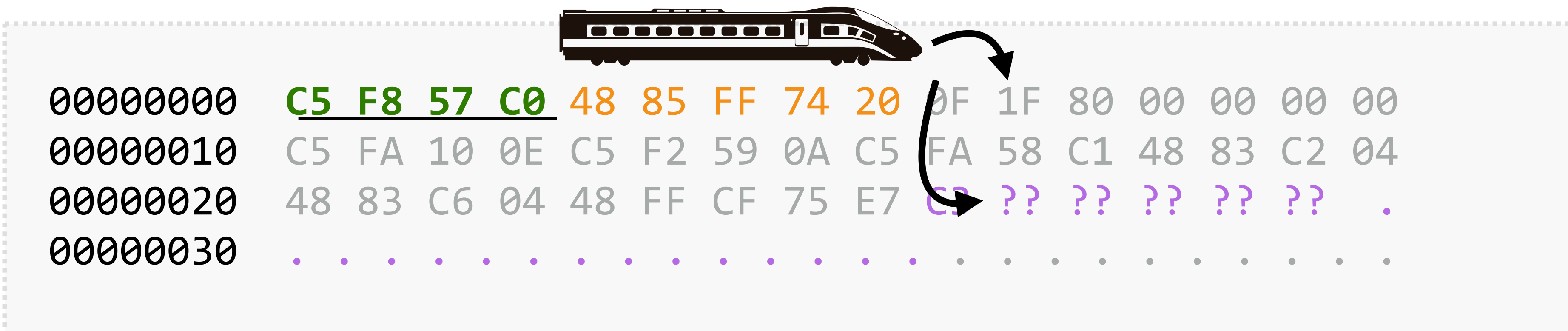


**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.



# The CPU seen as a high-speed train

To keep the pipeline fed, processor must look far ahead



**Executing** C5 F8 57 C0:

```
xorps xmm0, xmm0, xmm0
```

**Decoding**

```
0F 1F 80 00 00
```

```
00 00 C5 FA 10 0E:
```

**Scheduling** 48 85 FF 74 20:

```
test rdi, rdi
jz loop_end
```

???????

???????

???????



**MidJourney:** High speed train driving along a rail track. The driver in the cockpit looks through binoculars to observe obstacles far ahead.



# Branch prediction

- Every time your program contains a branch instruction, processor must decide what to do a long time in advance!
  - *To jump, or not to jump?* Waiting for the actual answer would be **too slow**.

# Branch prediction

- Every time your program contains a branch instruction, processor must decide what to do a long time in advance!
  - *To jump, or not to jump?* Waiting for the actual answer would be **too slow**.
- **The Algorithm:**
  - Ask the oracle: jump or not?
  - Go that way & hope it's correct! Begin executing the instructions.
  - N clock cycles later, we will know if it was the right decision.
    - Correct? All good. 😊

# Branch prediction

- Every time your program contains a branch instruction, processor must decide what to do a long time in advance!
  - *To jump, or not to jump?* Waiting for the actual answer would be **too slow**.
- **The Algorithm:**
  - Ask the oracle: jump or not?
  - Go that way & hope it's correct! Begin executing the instructions.
  - N clock cycles later, we will know if it was the right decision.
    - Correct? All good. 😊
    - Wrong? 😱 PANIC!!! Let's pull the emergency break, and undo all steps.



# Branch prediction

- Every time your program contains a branch instruction, processor must decide what to do a long time in advance!
  - *To jump, or not to jump?* Waiting for the actual answer would be **too slow**.
- **The Algorithm:**
  - Ask the oracle: jump or not?
  - Go that way & hope it's correct! Begin executing the instructions.
  - N clock cycles later, we will know if it was the right decision.
    - Correct? All good. 😊
    - Wrong? 😱 PANIC!!! Let's pull the emergency break, and undo all steps.



In our naïve implementation, this happens all the time!  
Values are random & positive with 50% probability...



```
# Generate a huge input array of random numbers (4 gigabytes)
n = 1024*1024*1024
x = np.float32(np.random.randn(n))
```

# Improved version

```
float psum_0(float *x, size_t size) {  
    float r { 0.f };  
  
    for (size_t i = 0; i < size; ++i) {  
        if (x[i] >= 0) {  
            r += x[i];  
            if (std::isinf(r))  
                abort();  
        }  
    }  
  
    return r;  
}
```

# Improved version

```
float psum_0(float *x, size_t size) {  
    float r { 0.f };  
  
    for (size_t i = 0; i < size; ++i) {  
        if (x[i] >= 0) {  
            r += x[i];  
            if (std::isinf(r))  
                abort();  
        }  
    }  
  
    return r;  
}
```

# Improved version

```
float psum_0(float *x, size_t size) {  
    float r { 0.f };
```

```
    for (size_t i = 0; i < size; ++i) {  
        if (x[i] >= 0) {  
            r += x[i];  
            if (std::isinf(r))  
                abort();  
        }  
    }
```

```
    return r;
```

```
}
```



```
    for (size_t i = 0; i < size; ++i)  
        r += fmaxf(x[i], 0.f);
```

**Demo time**

# Disassembly of psum\_1

```
; __int64 __fastcall psum_1(float *, unsigned __int64)
EXPORT __Z6psum_1Pfm
__Z6psum_1Pfm
CBZ X1, loc_36E8 ; Compare and Branch on Zero
```

```
MOVI D1, #0 ; Move Immediate
MOVI D0, #0 ; Move Immediate
```

```
loc_36E8
MOVI D0, #0 ; Move Immediate
RET ; Return from Subroutine
; End of function psum_1(float *,ulong)
```

```
loc_36D0
LDR S2, [X0], #4 ; Load from Memory
FMAXNM S2, S2, S1 ; Floating-point maxNum()
FADD S0, S0, S2 ; Floating-point Add
SUBS X1, X1, #1 ; Rd = Op1 - Op2
B.NE loc_36D0 ; Branch
```

```
RET ; Return from Subroutine
```

# Disassembly of psum\_1

```
loc_36D0
LDR      S2, [X0], #4 ; Load from Memory
FMAXNM  S2, S2, S1 ; Floating-point maxNum()
FADD    S0, S0, S2 ; Floating-point Add
SUBS    X1, X1, #1 ; Rd = Op1 - Op2
B.NE   loc_36D0 ; Branch
```

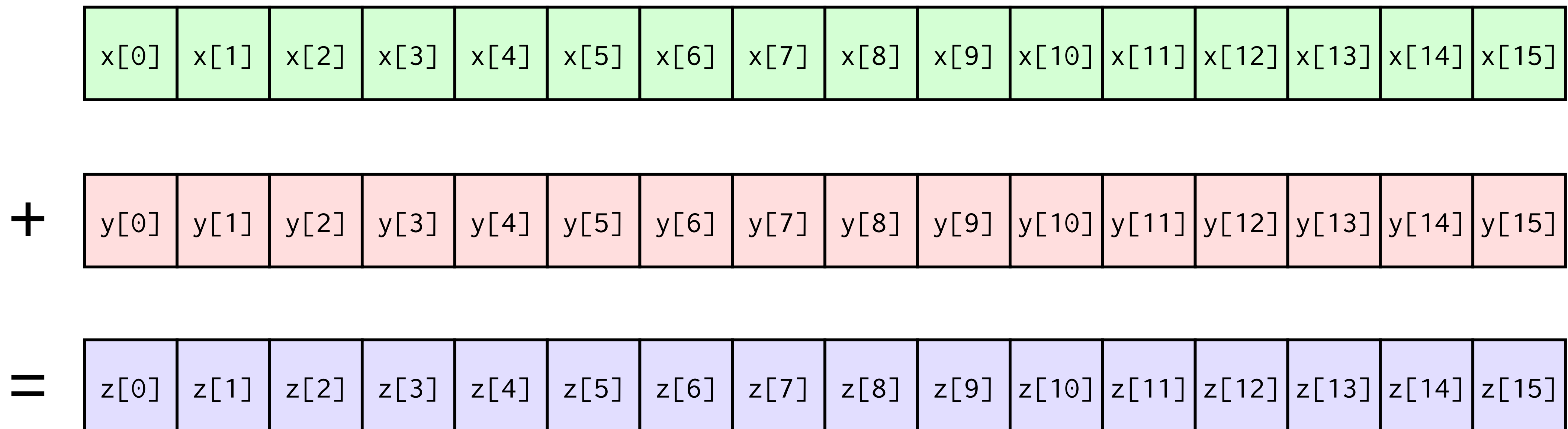
# CPUs can process vectors

This is known as **SIMD** (single instruction, multiple data)

In the same amount of time, can do either ...

$$\boxed{x[0]} + \boxed{y[0]} = \boxed{z[0]}$$

... Or ...



# Intrinsics

The screenshot shows the Intel® Intrinsic Guide website. The page title is "Intel® Intrinsic Guide". Below the title, there is a navigation menu with "Instruction Set" and "Categories". The "Instruction Set" menu includes MMX, SSE family, AVX family, AVX-512 family, AMX family, SVML, and Other. The "Categories" menu includes Application-Targeted, Arithmetic, Bit Manipulation, Cast, Compare, Convert, Cryptography, Elementary Math Functions, General Support, Load, Logical, Mask, Miscellaneous, Move, OS-Targeted, Probability/Statistics, Random, Set, Shift, Special Math Functions, Store, String Compare, Swizzle, Trigonometry, and Unknown. The main content area displays a list of intrinsics, each with its name, parameters, and return type. The intrinsics listed include `void __mm_2intersect_epi32`, `void __mm256_2intersect_epi32`, `void __mm512_2intersect_epi32`, `void __mm_2intersect_epi64`, `void __mm256_2intersect_epi64`, `void __mm512_2intersect_epi64`, `__m512i __mm512_4dpwssd_epi32`, `__m512i __mm512_mask_4dpwssd_epi32`, `__m512i __mm512_maskz_4dpwssd_epi32`, `__m512i __mm512_4dpwssds_epi32`, `__m512i __mm512_mask_4dpwssds_epi32`, `__m512i __mm512_maskz_4dpwssds_epi32`, `__m512 __mm512_4fmadd_ps`, `__m512 __mm512_mask_4fmadd_ps`, `__m512 __mm512_maskz_4fmadd_ps`, `__m128 __mm_4fmadd_ss`, `__m128 __mm_mask_4fmadd_ss`, `__m128 __mm_maskz_4fmadd_ss`, `__m512 __mm512_4fnmadd_ps`, `__m512 __mm512_mask_4fnmadd_ps`, `__m512 __mm512_maskz_4fnmadd_ps`, `__m128 __mm_4fnmadd_ss`, `__m128 __mm_mask_4fnmadd_ss`, `__m128 __mm_maskz_4fnmadd_ss`, `void __aadd_i32`, `void __aadd_i64`, `void __aand_i32`, `void __aand_i64`, and `__m128i __mm_abs_epi16`. The page also includes a search bar, a version number (3.6.6), a last updated date (05/10/2023), and a link to "Release Notes".

Intel & AMD SSE, AVX, AVX512— Intrinsic Explorer ([link](#)) ARM: NEON ([link](#)), SVE (datacenter processors, [link](#))

# Intrinsics

The screenshot shows the Intel® Intrinsic Guide website. The page title is "Intel® Intrinsic Guide" and it includes a navigation menu on the left with categories like "Instruction Set" (MMX, SSE family, AVX family, AVX-512 family, AMX family, SVML, Other) and "Categories" (Application-Targeted, Arithmetic, Bit Manipulation, Cast, Compare, Convert, Cryptography, Elementary Math Functions, General Support, Load, Logical, Mask, Miscellaneous, Move, OS-Targeted, Probability/Statistics, Random, Set, Shift, Special Math Functions, Store, String Compare, Swizzle, Trigonometry, Unknown). The main content area displays a list of intrinsics, each with its signature and a corresponding intrinsic name. The list includes functions like `void __mm_2intersect_epi32`, `void __mm256_2intersect_epi32`, `void __mm512_2intersect_epi32`, `void __mm_2intersect_epi64`, `void __mm256_2intersect_epi64`, `void __mm512_2intersect_epi64`, `__m512i __mm512_4dpwssd_epi32`, `__m512i __mm512_mask_4dpwssd_epi32`, `__m512i __mm512_maskz_4dpwssd_epi32`, `__m512i __mm512_4dpwssds_epi32`, `__m512i __mm512_mask_4dpwssds_epi32`, `__m512i __mm512_maskz_4dpwssds_epi32`, `__m512 __mm512_4fmadd_ps`, `__m512 __mm512_mask_4fmadd_ps`, `__m512 __mm512_maskz_4fmadd_ps`, `__m128 __mm_4fmadd_ss`, `__m128 __mm_mask_4fmadd_ss`, `__m128 __mm_maskz_4fmadd_ss`, `__m512 __mm512_4fnmadd_ps`, `__m512 __mm512_mask_4fnmadd_ps`, `__m512 __mm512_maskz_4fnmadd_ps`, `__m128 __mm_4fnmadd_ss`, `__m128 __mm_mask_4fnmadd_ss`, `__m128 __mm_maskz_4fnmadd_ss`, `void __aadd_i32`, `void __aadd_i64`, `void __aand_i32`, `void __aand_i64`, and `__m128i __mm_abs_epi16`. The page also includes a search bar, a version number (3.6.6), a last updated date (05/10/2023), and a link to "Release Notes".

Intel & AMD SSE, AVX, AVX512— Intrinsic Explorer ([link](#)) ARM: NEON ([link](#)), SVE (datacenter processors, [link](#))

# Disassembly of psum\_2

```
;__int64 __fastcall psum_2(float *, unsigned __int64)
EXPORT __Z6psum_2Pfm
__Z6psum_2Pfm

var_10= -0x10

SUB      SP, SP, #0x10 ; Rd = Op1 - Op2
MOVI     V0.2D, #0 ; Move Immediate
CMP      X1, #4 ; Set cond. codes on Op1 - Op2
B.CC     loc_3720 ; Branch
```

```
LSR      X8, X1, #2 ; Logical Shift Right
MOVI     V1.2D, #0 ; Move Immediate
MOVI     V0.2D, #0 ; Move Immediate
```

```
loc_370C
LDR      Q2, [X0],#0x10 ; Load from Memory
FMAX     V2.4S, V2.4S, V1.4S ; Floating-point Maximum
FADD     V0.4S, V0.4S, V2.4S ; Floating-point Add
SUBS     X8, X8, #1 ; Rd = Op1 - Op2
B.NE     loc_370C ; Branch
```

```
loc_3720
STR      Q0, [SP,#0x10+var_10] ; Store to Memory
MOV      W8, #4 ; Rd = Op2
MOV      X9, SP ; Rd = Op2
```

```
loc_372C
LDR      S1, [X9,X8] ; Load from Memory
FADD     S0, S1, S0 ; Floating-point Add
ADD      X8, X8, #4 ; Rd = Op1 + Op2
CMP      X8, #0x10 ; Set cond. codes on Op1 - Op2
B.NE     loc_372C ; Branch
```

```
ADD      SP, SP, #0x10 ; Rd = Op1 + Op2
RET      ; Return from Subroutine
; End of function psum_2(float *,ulong)
```

# Disassembly of psum\_2

```
loc_370C
LDR      Q2, [X0],#0x10 ; Load from Memory
FMAX    V2.4S, V2.4S, V1.4S ; Floating-point Maximum
FADD    V0.4S, V0.4S, V2.4S ; Floating-point Add
SUBS    X8, X8, #1 ; Rd = Op1 - Op2
B.NE    loc_370C ; Branch
```

# Disassembly of psum\_2

```
loc_370C
LDR      Q2, [X0],#0x10 ; Load from Memory
FMAX    V2.4S, V2.4S, V1.4S ; Floating-point Maximum
FADD    V0.4S, V0.4S, V2.4S ; Floating-point Add
SUBS    X8, X8, #1 ; Rd = Op1 - Op2
B.NE    loc_370C ; Branch
```

The idea was always that compilers would magically insert vector instructions for us. ("auto-vectorization")  
**Sad part:** is never really worked, still does not work well enough.

**Demo time**

# What's missing?

# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.

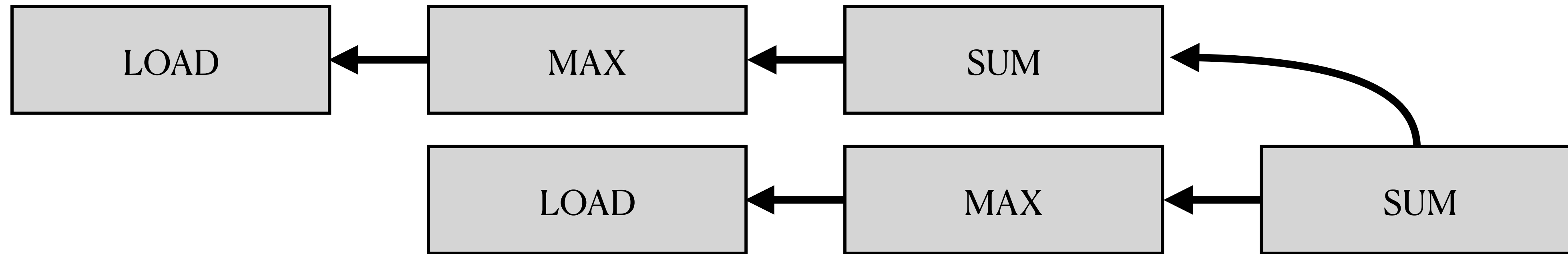
# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



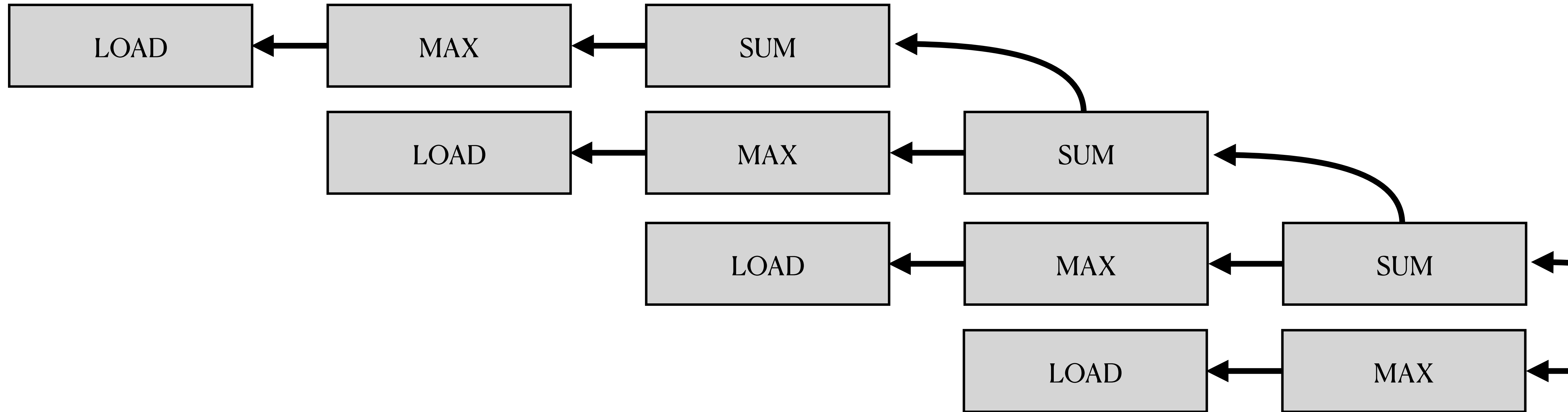
# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



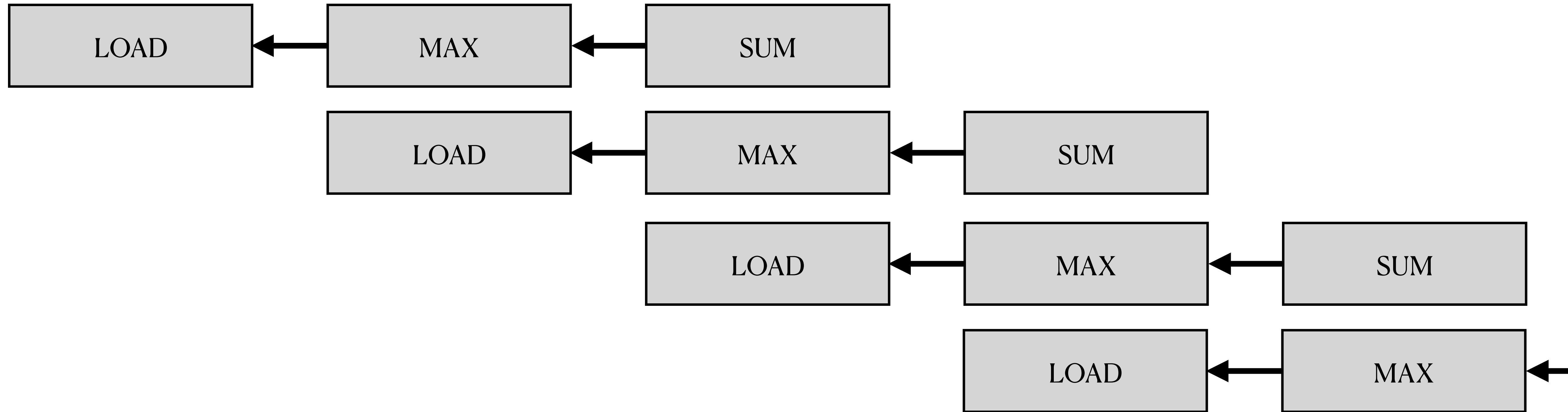
# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



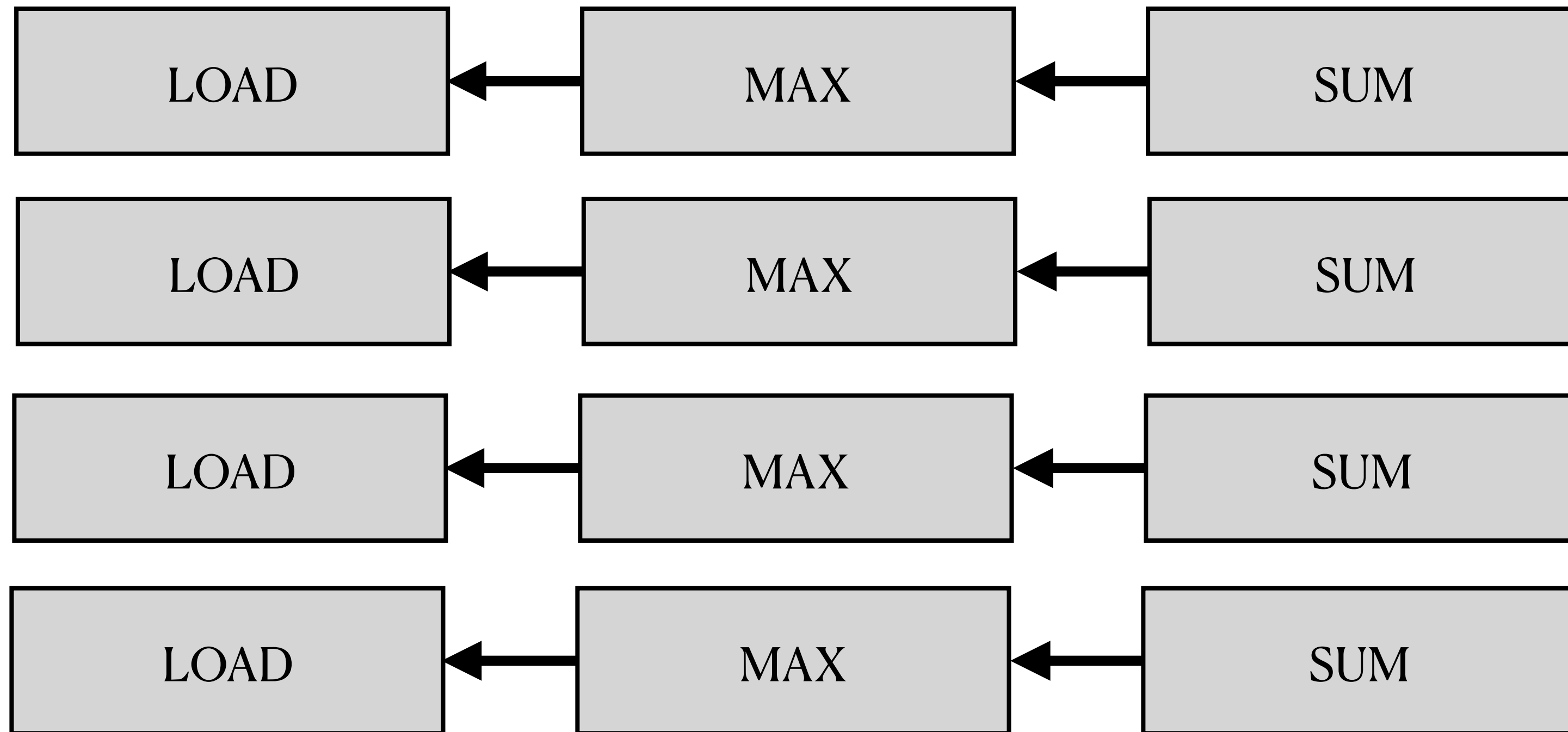
# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



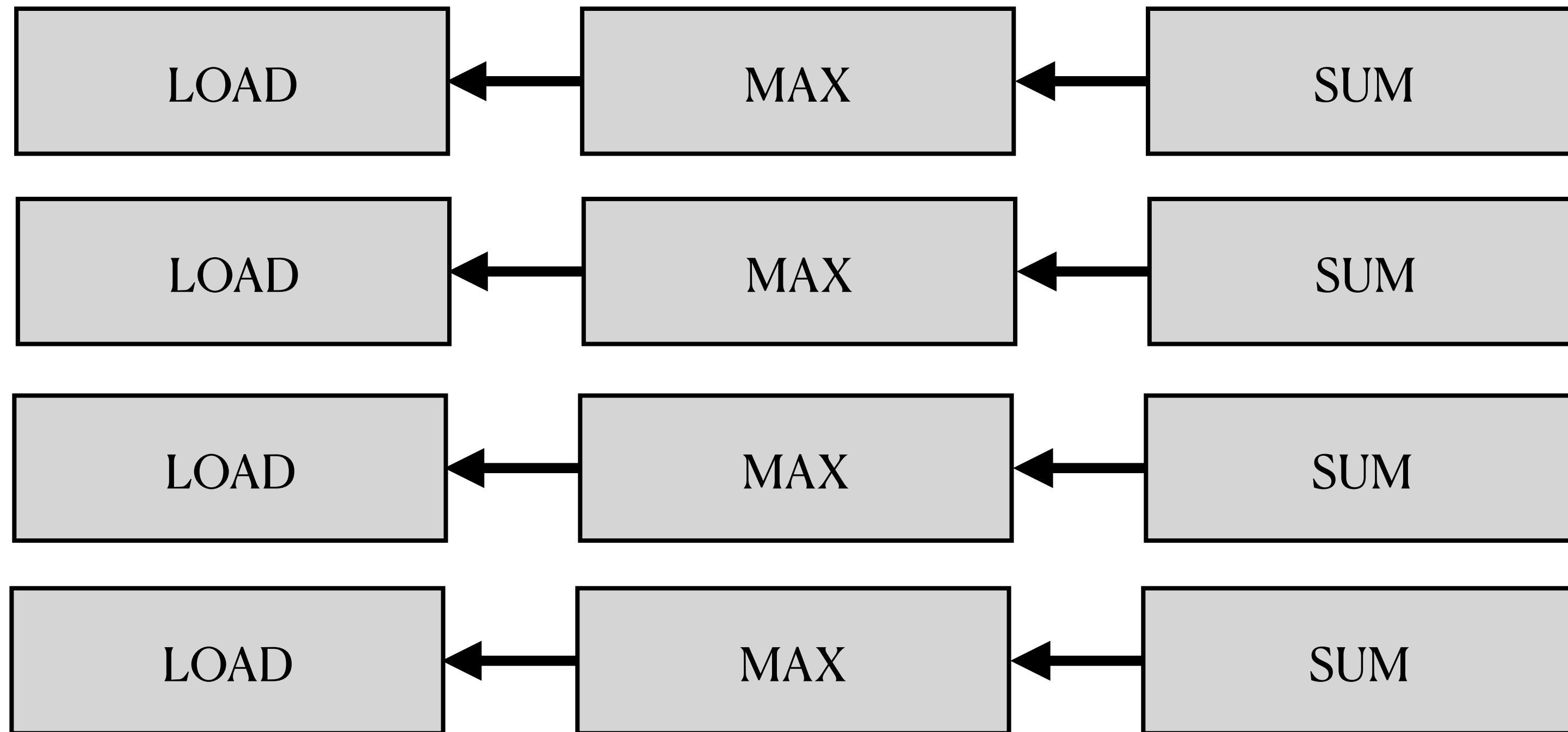
# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



# What's missing?

- Our program is still not parallel enough. Can't keep the pipeline fed.



- Break those troublesome connections. We can compute 4 (or 8 or 16) partial sums and then add them up at the end.

# Disassembly of psum\_3

```
;__int64 __fastcall psum_3(float *, unsigned __int64)
EXPORT __Z6psum_3Pfm
__Z6psum_3Pfm

var_10= -0x10

SUB     SP, SP, #0x10 ; Rd = Op1 - Op2
MOVI   V0.2D, #0 ; Move Immediate
CMP    X1, #0x10 ; Set cond. codes on Op1 - Op2
B.CC   loc_37BC ; Branch
```

```
LSR    X8, X1, #4 ; Logical Shift Right
MOVI   V0.2D, #0 ; Move Immediate
MOVI   V1.2D, #0 ; Move Immediate
MOVI   V2.2D, #0 ; Move Immediate
MOVI   V3.2D, #0 ; Move Immediate
MOVI   V4.2D, #0 ; Move Immediate
```

```
loc_377C
LDP    Q5, Q6, [X0] ; Load Pair
LDP    Q7, Q16, [X0, #0x20] ; Load Pair
FMAX   V5.4S, V5.4S, V0.4S ; Floating-point Maximum
FADD   V1.4S, V1.4S, V5.4S ; Floating-point Add
FMAX   V5.4S, V6.4S, V0.4S ; Floating-point Maximum
FADD   V2.4S, V2.4S, V5.4S ; Floating-point Add
FMAX   V5.4S, V7.4S, V0.4S ; Floating-point Maximum
FADD   V3.4S, V3.4S, V5.4S ; Floating-point Add
FMAX   V5.4S, V16.4S, V0.4S ; Floating-point Maximum
FADD   V4.4S, V4.4S, V5.4S ; Floating-point Add
ADD    X0, X0, #0x40 ; '@' ; Rd = Op1 + Op2
SUBS   X8, X8, #1 ; Rd = Op1 - Op2
B.NE   loc_377C ; Branch
```

```
FADD   V0.4S, V2.4S, V1.4S ; Floating-point Add
FADD   V1.4S, V4.4S, V3.4S ; Floating-point Add
FADD   V0.4S, V1.4S, V0.4S ; Floating-point Add
```

```
loc_37BC
STR    Q0, [SP, #0x10+var_10] ; Store to Memory
MOV    W8, #4 ; Rd = Op2
MOV    X9, SP ; Rd = Op2
```

```
loc_37C8
LDR    S1, [X9, X8] ; Load from Memory
FADD   S0, S1, S0 ; Floating-point Add
ADD    X8, X8, #4 ; Rd = Op1 + Op2
CMP    X8, #0x10 ; Set cond. codes on Op1 - Op2
B.NE   loc_37C8 ; Branch
```

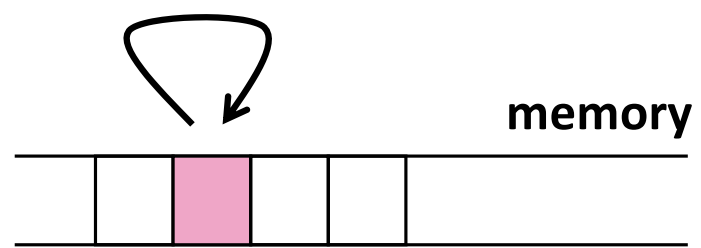
```
ADD    SP, SP, #0x10 ; Rd = Op1 + Op2
RET    ; Return from Subroutine
; End of function psum_3(float *,ulong)
```

# Disassembly of psum\_3

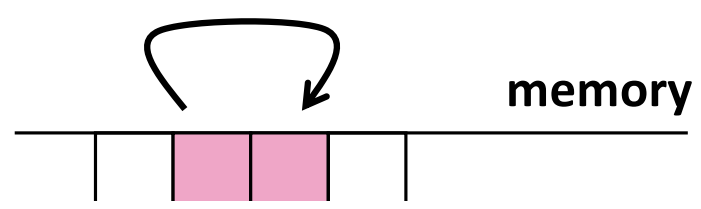
```
loc_377C
LDP      Q5, Q6, [X0] ; Load Pair
LDP      Q7, Q16, [X0, #0x20] ; Load Pair
FMAX     V5.4S, V5.4S, V0.4S ; Floating-point Maximum
FADD     V1.4S, V1.4S, V5.4S ; Floating-point Add
FMAX     V5.4S, V6.4S, V0.4S ; Floating-point Maximum
FADD     V2.4S, V2.4S, V5.4S ; Floating-point Add
FMAX     V5.4S, V7.4S, V0.4S ; Floating-point Maximum
FADD     V3.4S, V3.4S, V5.4S ; Floating-point Add
FMAX     V5.4S, V16.4S, V0.4S ; Floating-point Maximum
FADD     V4.4S, V4.4S, V5.4S ; Floating-point Add
ADD      X0, X0, #0x40 ; '@' ; Rd = Op1 + Op2
SUBS     X8, X8, #1 ; Rd = Op1 - Op2
B.NE     loc_377C ; Branch
```

# Memory & Caches

- Load instruction  
**Latency:** ~5-300 cycles  
**Throughput:** 2
- Depends on where we go in the memory hierarchy.

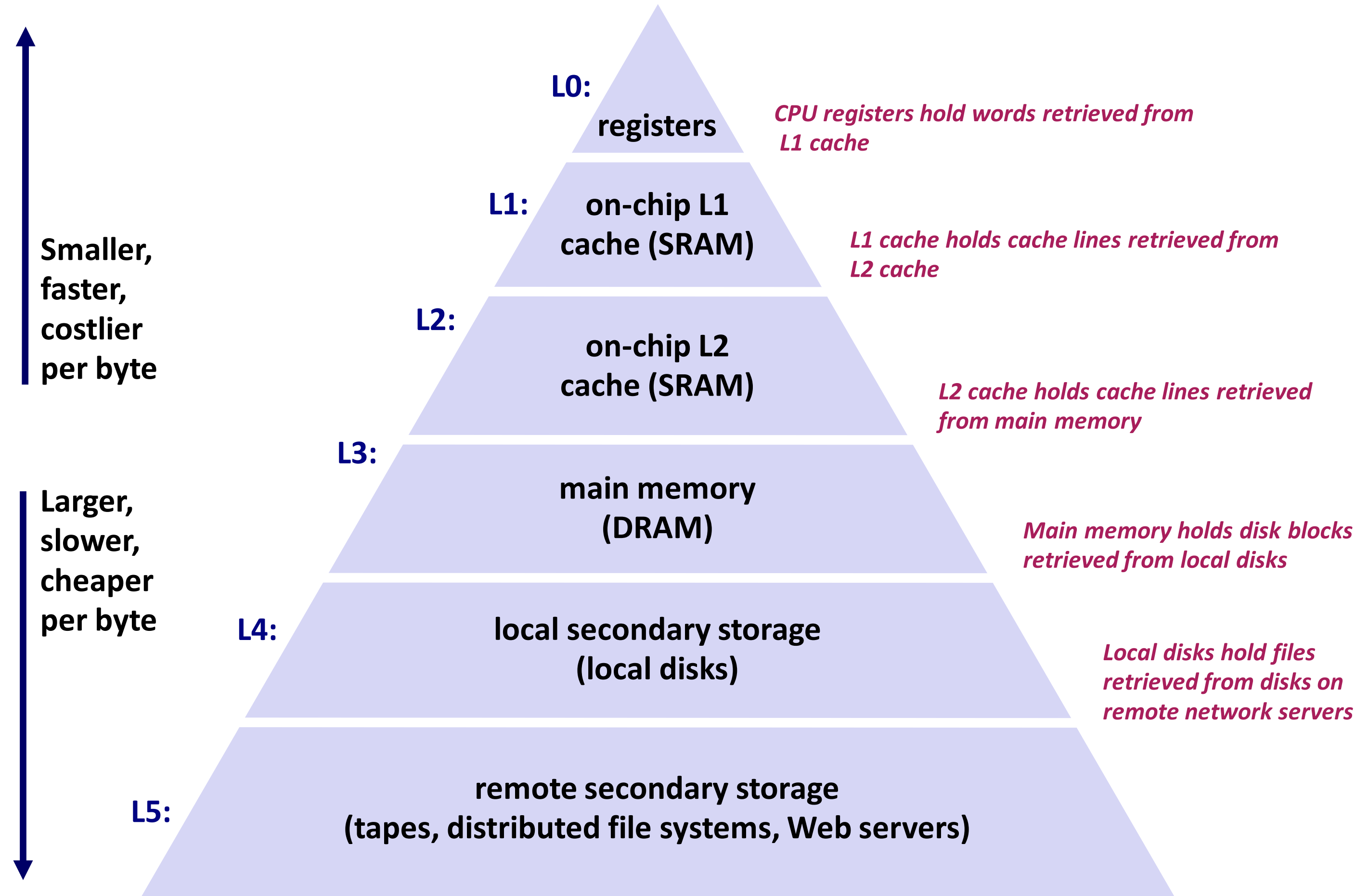


Temporal locality



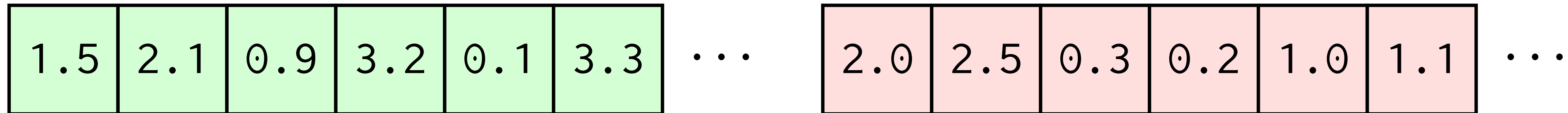
Spatial locality

## Typical Memory Hierarchy

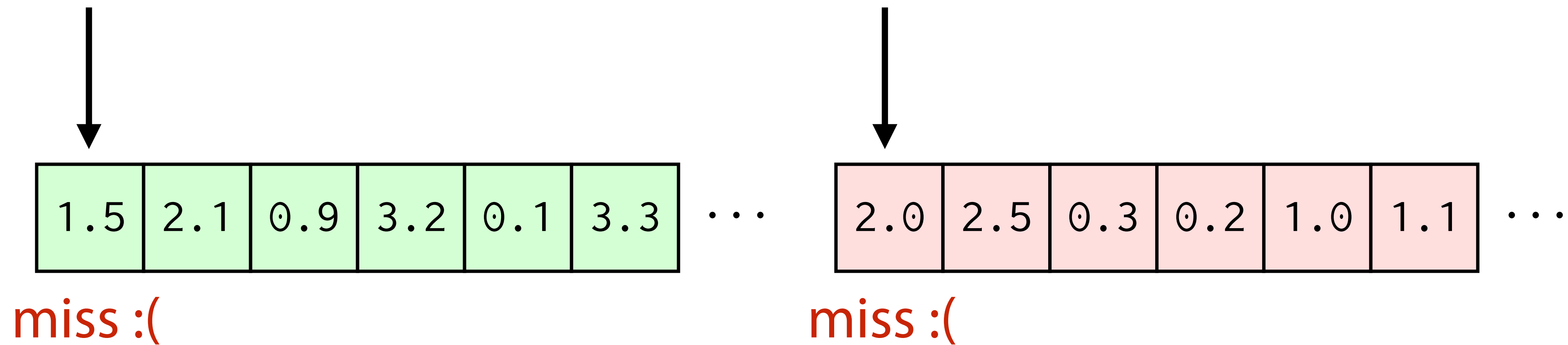


[Markus Püschel, ETHZ]

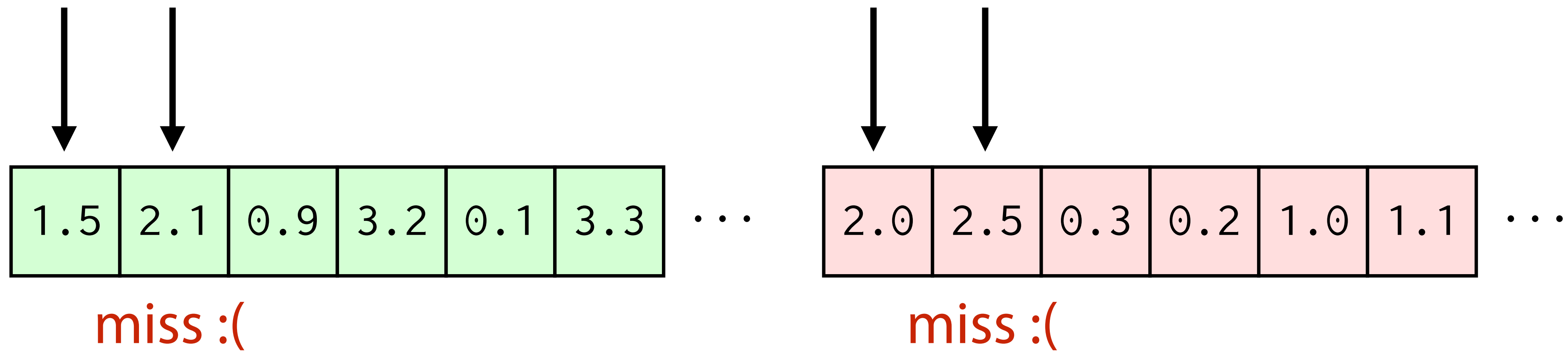
# Prefetchers exploit spatial & temporal regularity



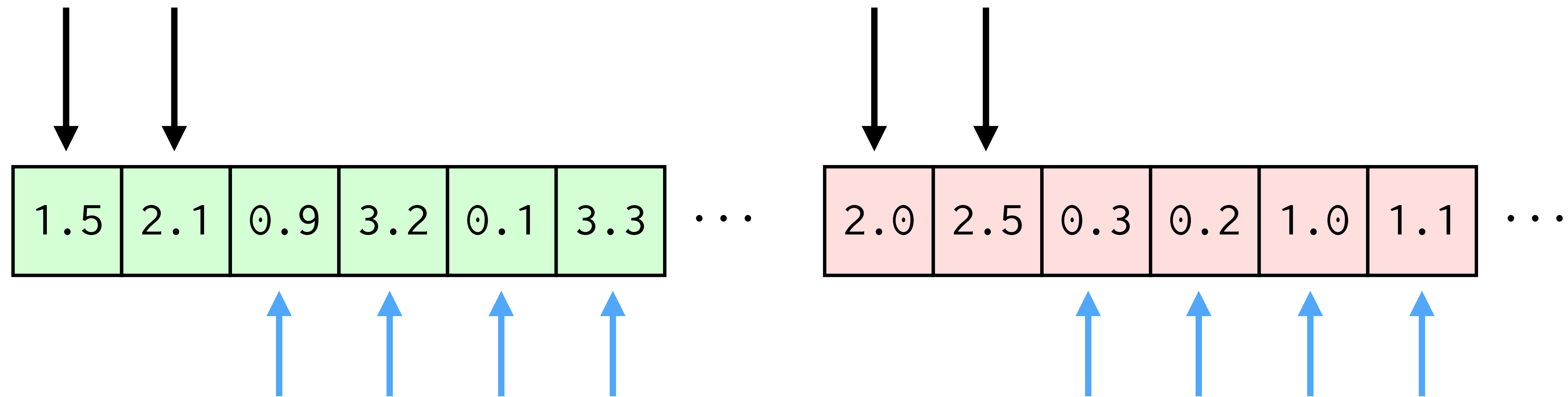
# Prefetchers exploit spatial & temporal regularity



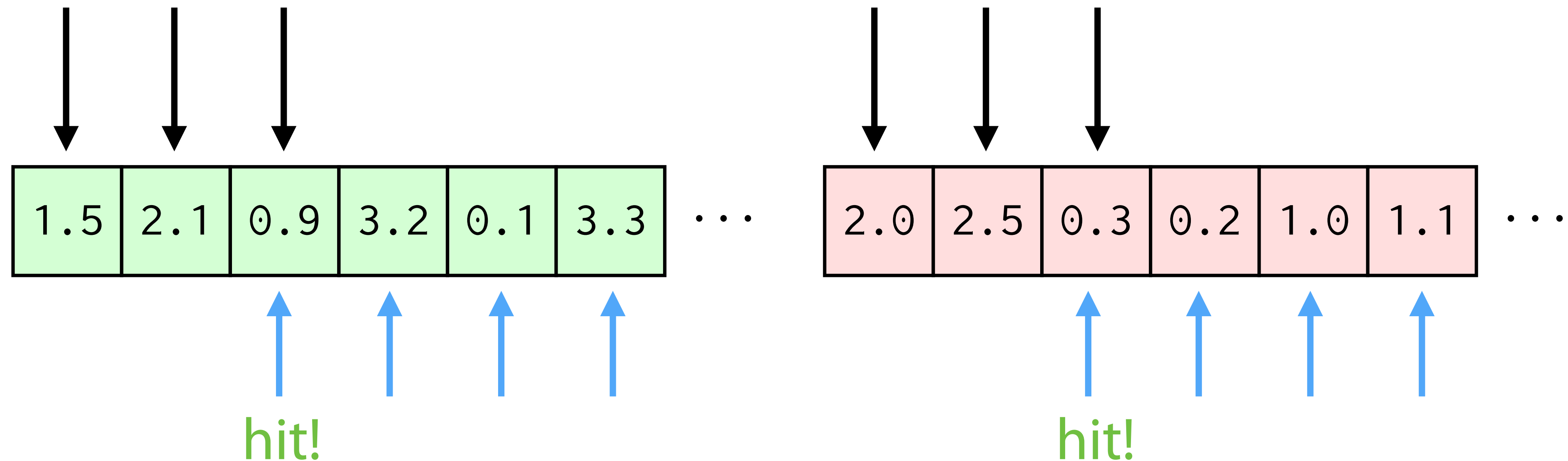
# Prefetchers exploit spatial & temporal regularity



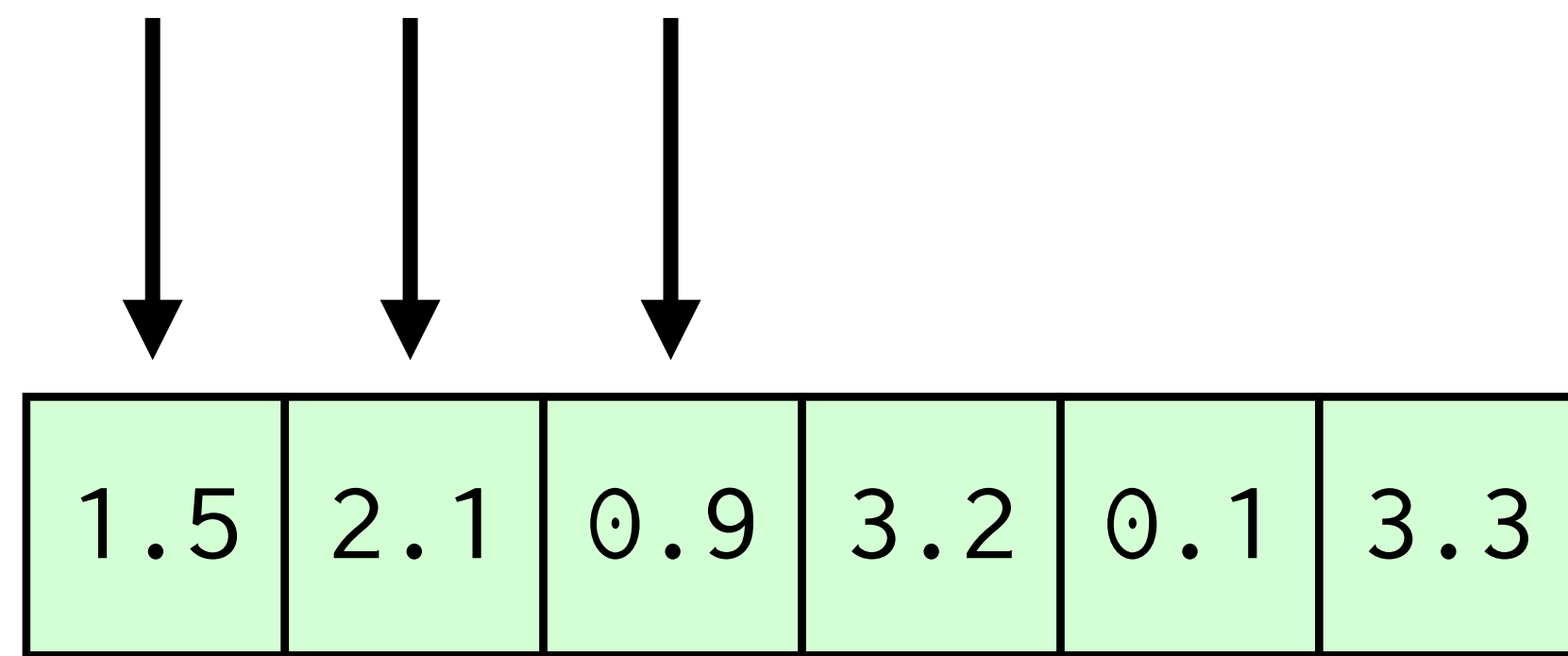
# Prefetchers exploit spatial & temporal regularity



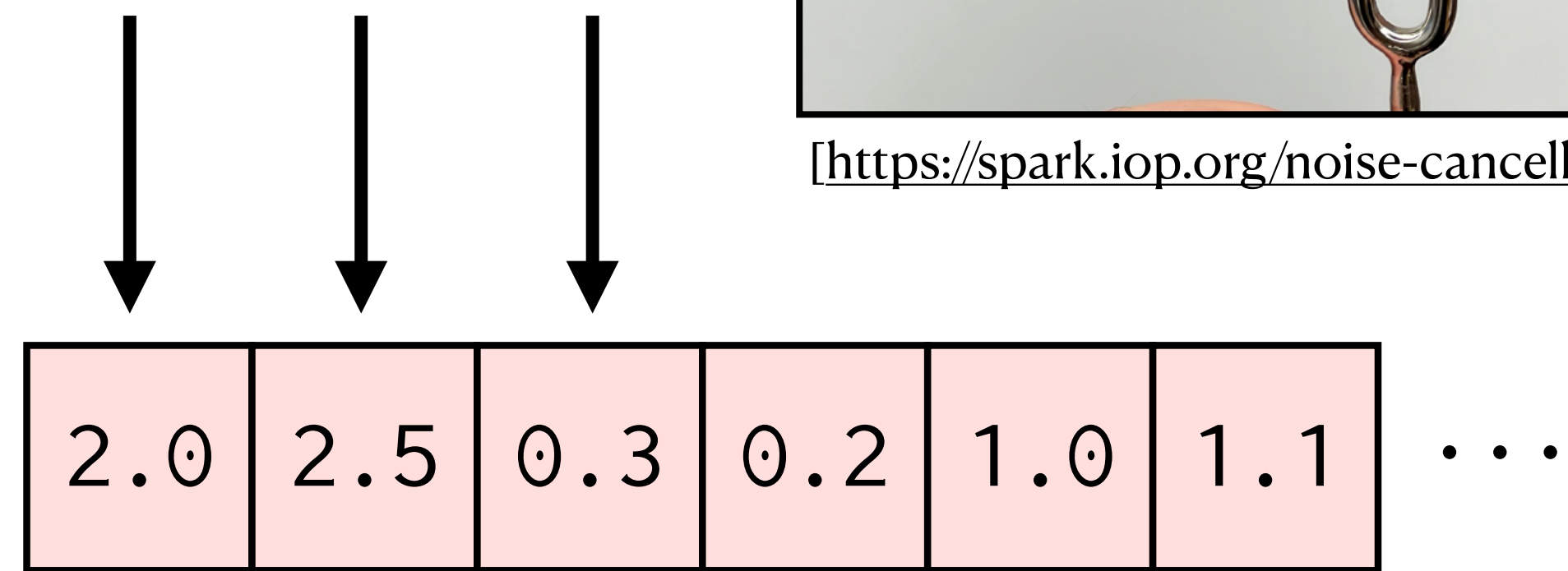
# Prefetchers exploit spatial & temporal regularity



# Prefetchers exploit spatial & temporal regularity



hit!



hit!



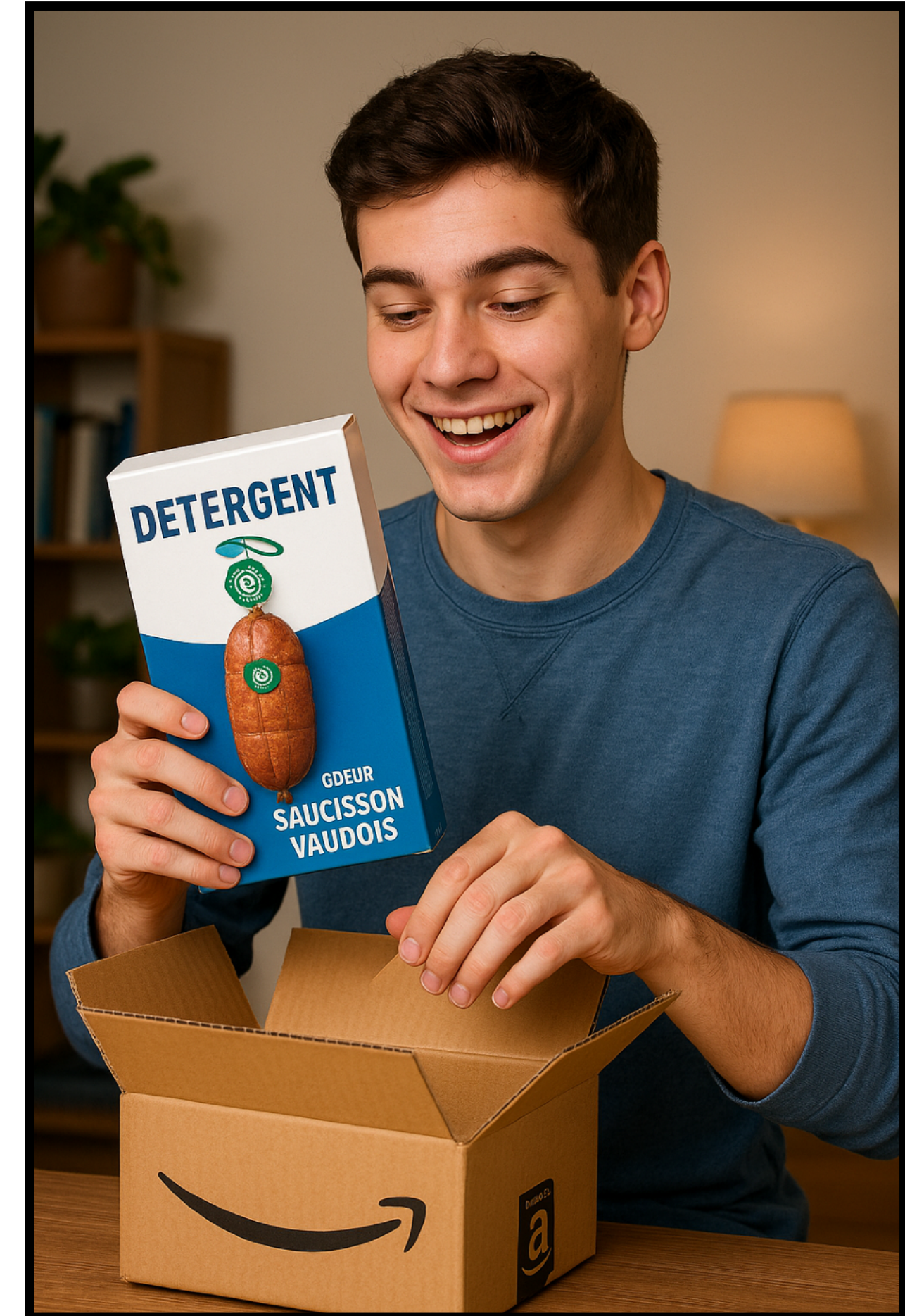
[<https://spark.iop.org/noise-cancelling-tuning-fork>]

a

[Images by ChatGPT]



Get something predictable from the corner store.



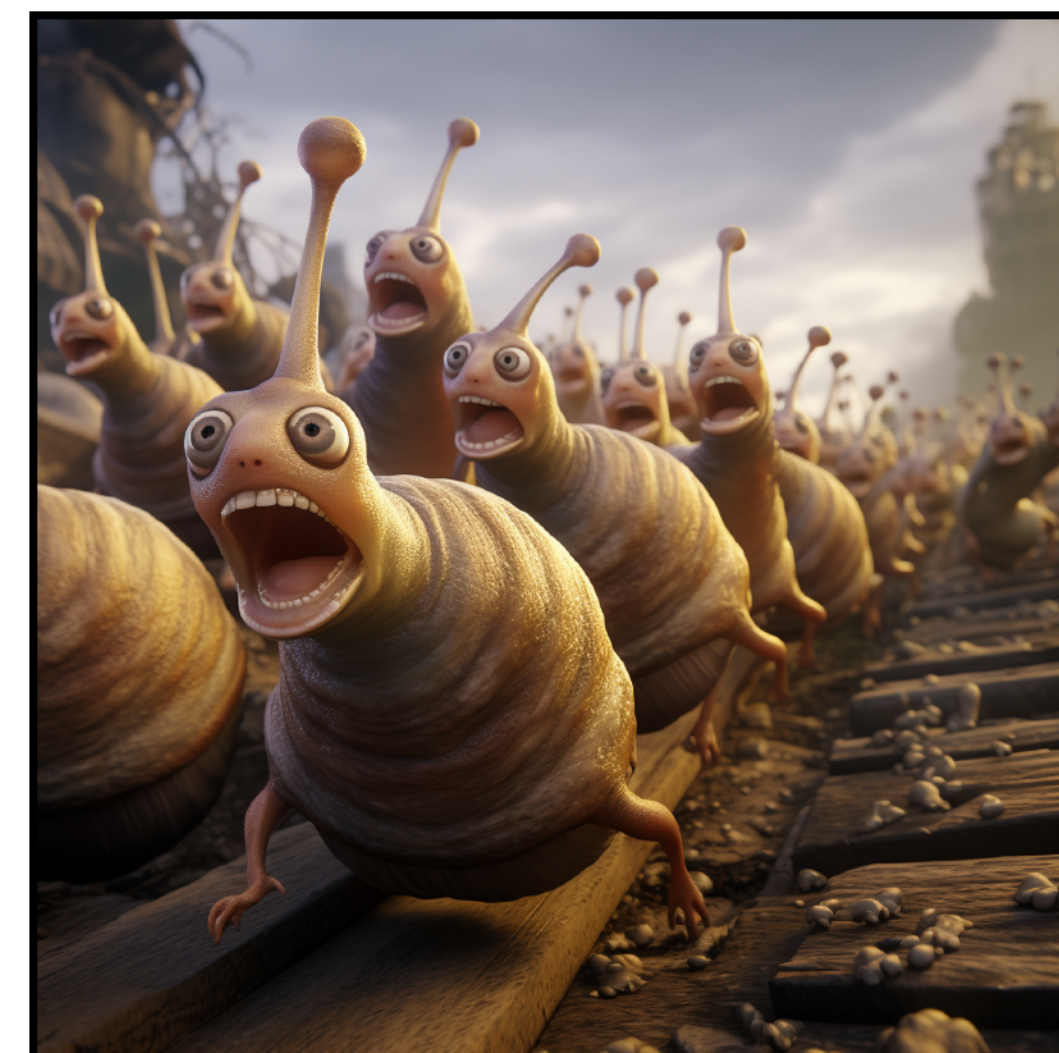
Weird unpredictable access, have to wait for a long time.

# CPU vs GPU

- All this fancy stuff
  - issuing multiple instructions per cycle
  - huge caches
  - branch prediction, speculative execution
- .. **GPUs don't have it!**
- If a CPU is a **cheetah** (a product of evolution designed to be super-fast), then a GPU is an **army of snails**.
- Each core much simpler & smaller, but also very slow. In exchange, can have **lots** of them.



**MidJourney:** a Cheetah



**MidJourney:** an army of snails charging forward

# General advice

- **When optimizing something, be wary of**
  - .. sacrificing correctness for speed
    - *Ensure correctness (e.g. via test framework) first*
  - .. reinventing the wheel
    - *Use right tools, libraries, etc.*
  - .. valuing computer time over your own time
    - *Maybe optimization isn't really needed.*
- **Always measure:** most time is spent in a few bottlenecks.  
Locate them and focus on just this part.

# The end.

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.



I AM A GOD.

[<https://xkcd.com/676>]