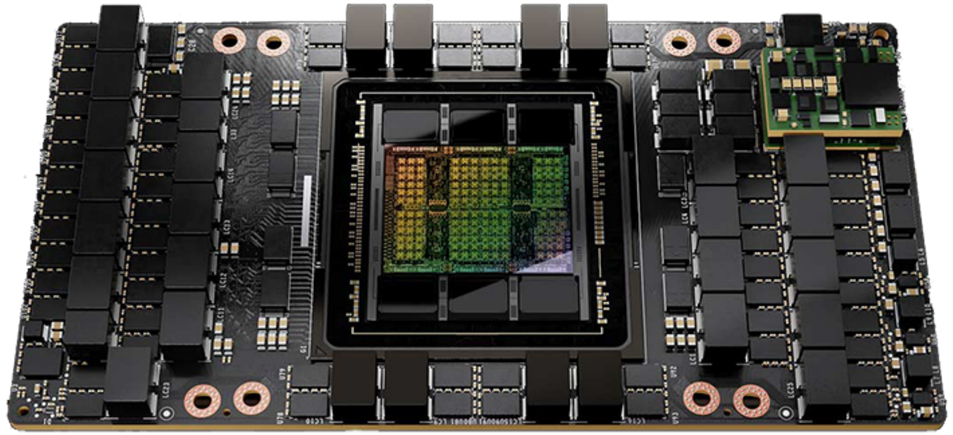
September 4, 2025

6.S894 Accelerated Computing Lecture 1: The Problem

Jonathan Ragan-Kelley

Hardware is **amazing** today

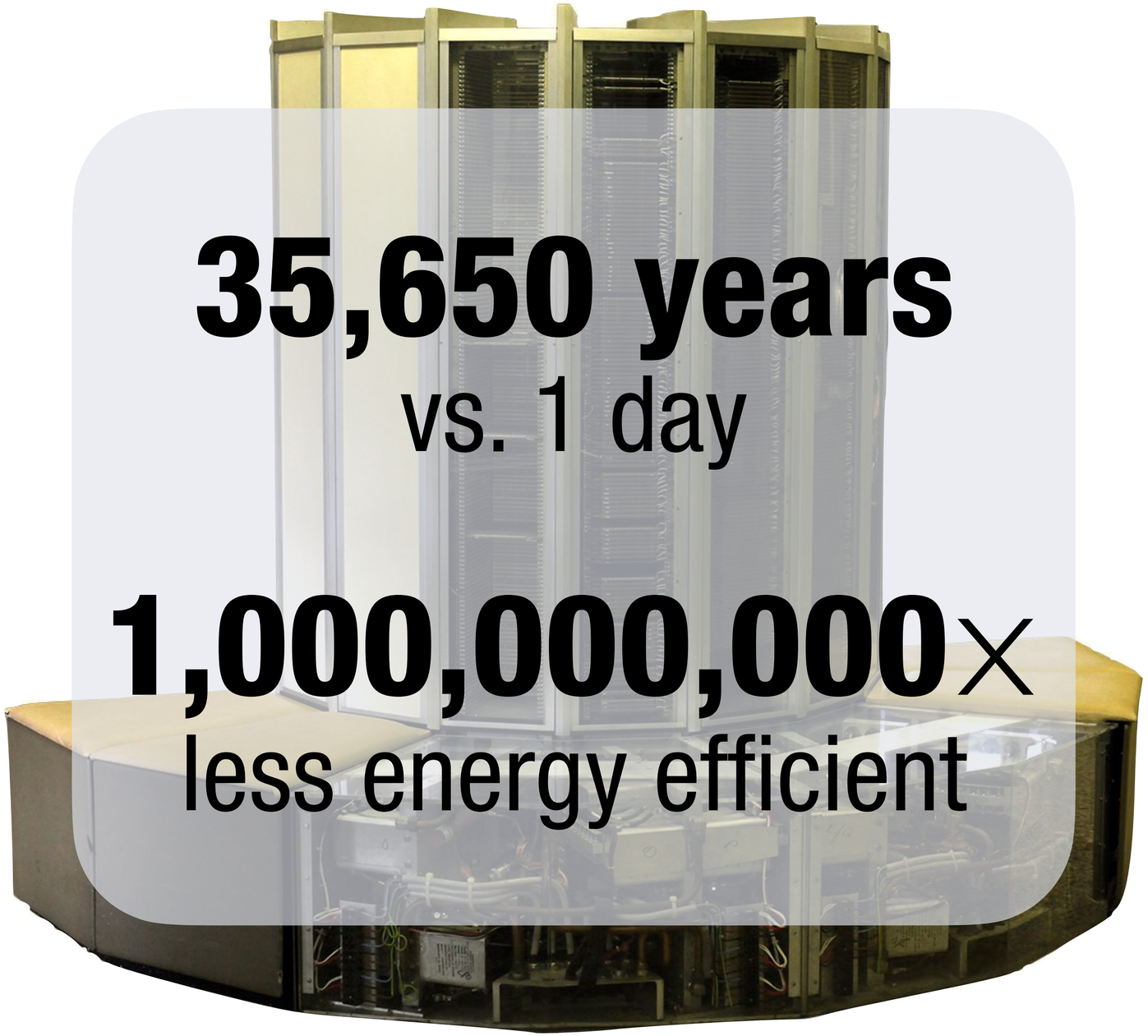
LLaMA by Meta



1B parameters
× 30B tokens
× 6 FLOPS/parameter

180 ExaFLOPS (16-bit)

1 day
on 8 GPUs



35,650 years
vs. 1 day

1,000,000,000×
less energy efficient

Cray 1 (1975)
the first “supercomputer”

160 MegaFLOPS
8MB RAM

115 kW

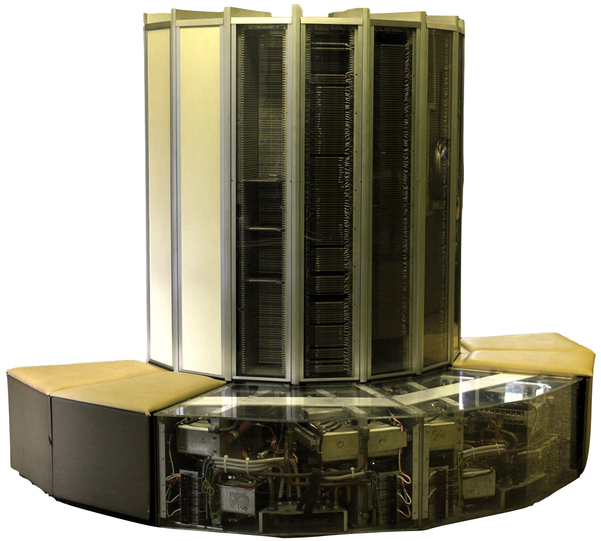
$7.9M
($56M in 2025 dollars)

50 years isn’t as long as it sounds

115 mph · 22 mpg

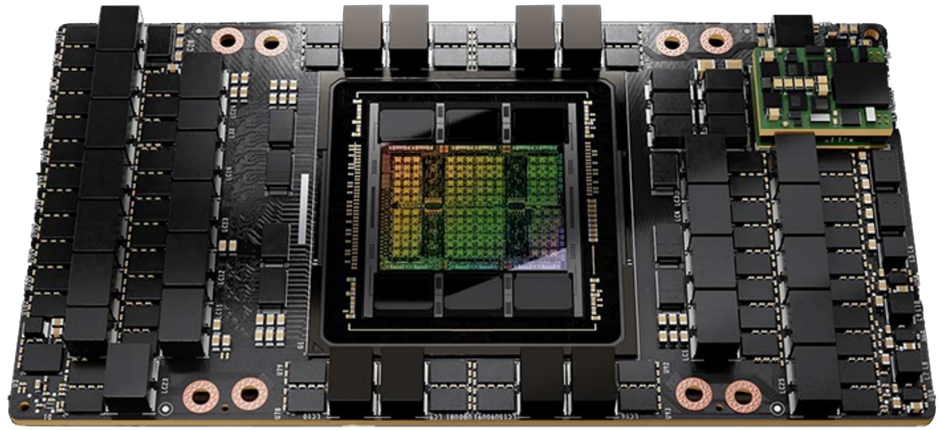
560 mph

1975



700 Watts / **Mega**FLOP

2024



700 Watts / **Peta**FLOP

1,000,000,000×

~20 **Tera**FLOPs

Let’s multiply some matrices

Python

```
# A,B,C : List[float]
# N = 2048
for i in range(N):
  for j in range(N):
    for k in range(N):
      C[i*N + j] += A[i*N + k] * B[k*N + j]
```

10 MegaFLOPS

0.00005% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

JavaScript

```
// A,B,C : Array<number>
// N = 2048
for (let i = 0; i < N; i++) {
  for (let j = 0; j < N; j++) {
    for (let k = 0; k < N; k++) {
      C[i*N + j] += A[i*N + k] * B[k*N + j];
}}}
```

500 MegaFLOPS

0.0025% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

C (clang18 -O3 -march=native …)

```
// float *A, *B, *C
// N = 2048
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    for (int k = 0; k < N; ++k) {
      C[i*N + j] += A[i*N + k] * B[k*N + j];
}}}
```

1 GigaFLOPS

0.005% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let's multiply some matrices

C (clang18 -O3 -march=native …)

```
// float *A, *B, *C
// N = 2048
for (int i = 0; i < N; ++i) {
  for (int k = 0; k < N; ++k) {
    for (int j = 0; j < N; ++j) {
      C[i*N + j] += A[i*N + k] * B[k*N + j];
}}}
```

20 GigaFLOPS

0.1% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

C (`clang18 -O3 -march=native …`)

+ hand-tuned,
vectorized,
cache blocked
(1 core)

40 GigaFLOPS

0.2% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

C (`clang18 -O3 -march=native …`)

+ hand-tuned,
vectorized,
cache blocked
+ parallel
(4+4 cores)

200 GigaFLOPS

1% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

C & Assembly (Apple `Accelerate`)

+ hand-tuned,
cache blocked
+ quantized to bf16,
+ matrix units

1.5 TeraFLOPS
7.5% of SoC peak

Apple M2

~20 **Tera**FLOPs

Let’s multiply some matrices

Metal compute shaders

```
static inline void accumulateOuterProduct(
    const device float4& inputA,
    const device float4& inputB,
    thread float4x4& output
)
{
    output[0] += inputA.x * inputB;
    output[1] += inputA.y * inputB;
    output[2] += inputA.z * inputB;
    output[3] += inputA.w * inputB;
}

/// Each thread of this kernel operates on a 8x8 sector of the
/// output buffer. Matrix buffers require padding to accomodate this.
///
/// Requirements:
///
/// The bytes-per-row of each matrix buffer must be padded to a multiple
/// of eight floats (32 bytes). Similarly, the row count of each matrix
/// must be padded to a multiple of eight.
kernel void MultiplyMatrices(
    constant BufferDimensions& dimensions [[ buffer(0) ]],

    const device float4* inputA [[ buffer(1) ]],
    const device float4* inputB [[ buffer(2) ]],
    device float4*       output [[ buffer(3) ]],

    ushort2 outputSector [[ thread_position_in_grid ]]
)
{
    const ushort2 outputPosition = outputSector * sectorSize;

    if (
        outputPosition.x >= dimensions.outputRowCount ||
        outputPosition.y >= dimensions.outputColumnCount
    )
    {
        return;
    }

    float4x4 s00 = float4x4(0.f), s01 = float4x4(0.f);
    float4x4 s10 = float4x4(0.f), s11 = float4x4(0.f);
    inputA += outputPosition.x / 4;
    inputB += outputPosition.y / 4;

    const Dimension strideA = dimensions.bytesPerRowA / sizeof(float4);
    const Dimension strideB = dimensions.bytesPerRowB / sizeof(float4);
    const device float4* const endOfB = inputB + dimensions.innerInputDimension * strideB;

    while (inputB < endOfB) {
        accumulateOuterProduct(inputA[0], inputB[0], s00);
        accumulateOuterProduct(inputA[0], inputB[1], s01);
        accumulateOuterProduct(inputA[1], inputB[0], s10);
        accumulateOuterProduct(inputA[1], inputB[1], s11);

        inputA += strideA;
        inputB += strideB;
    }

    const Dimension outputStride = dimensions.outputBytesPerRow / sizeof(float4);
    output += outputPosition.x * outputStride + outputPosition.y / 4;

    output[0] = s00[0]; output[1] = s01[0]; output += outputStride;
    output[0] = s00[1]; output[1] = s01[1]; output += outputStride;
    output[0] = s00[2]; output[1] = s01[2]; output += outputStride;
    output[0] = s00[3]; output[1] = s01[3]; output += outputStride;
    output[0] = s10[0]; output[1] = s11[0]; output += outputStride;
    output[0] = s10[1]; output[1] = s11[1]; output += outputStride;
    output[0] = s10[2]; output[1] = s11[2]; output += outputStride;
    output[0] = s10[3]; output[1] = s11[3];
}
```

1000s of lines…

Apple M2

~20 **Tera**FLOPs

3.2 **Tera**FLOPS

15% of SoC peak

Let’s multiply some matrices

Neural Engine

Proprietary ISA
& interface

int16 quantized

16 TeraOPS
80% of SoC peak

Apple M2

~20 **Tera**(FL)OPs

Version	% Peak
NPU	80%
GPU	15%
CPU Matrix	7.5%
Opt. Par. C	1%
Opt. C	0.2%
C	0.005%
JavaScript	0.0025%
Python	0.00005%

6 orders of magnitude performance & efficiency

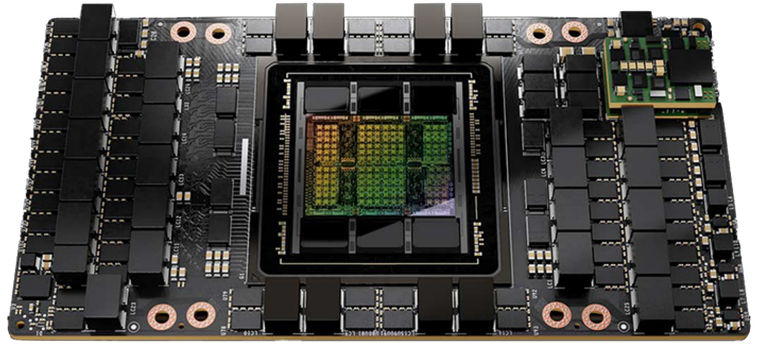
Extreme programming effort & specialization of code

Apple M2

~20 **Tera**(FL)OPs

Version	% Peak
NPU	80%
GPU	15%
CPU Ma ix	5%
Opt. Par.	1%
C C	0. %
C	0.005 %
avaScript	0.0025%
Python	0.00005%

NVIDIA H100



~1000 **Tera**FLOPs

Hardware is amazing, **but…**

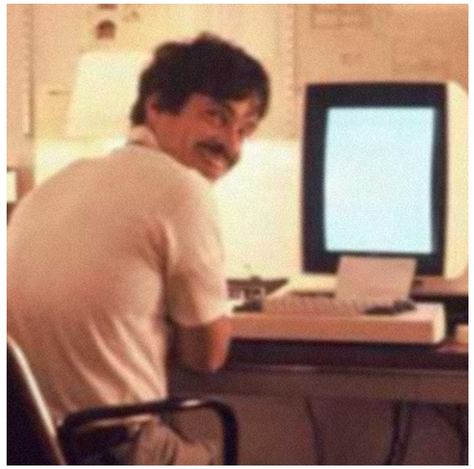
the **gap** between
reasonable code
and **peak performance**
is enormous.

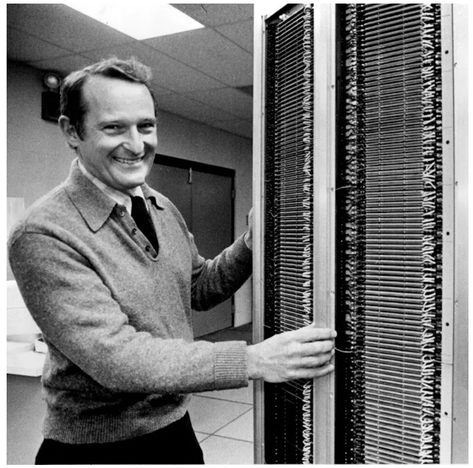
clean,
modular,
portable…

Why does this gap exist?

Hardware has **radically changed**

Software largely **hasn't**





1975 had:

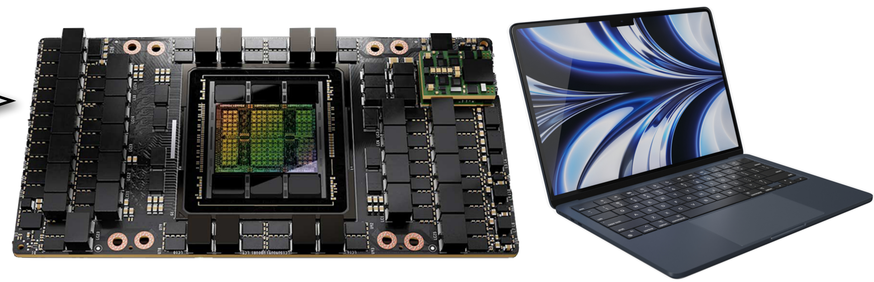
- UNIX
- C
- virtual memory
- optimizing compilers
- ML, type inference
- Smalltalk
- Networked GUI workstations

Increasing gap between **Programming Models** and **Hardware**

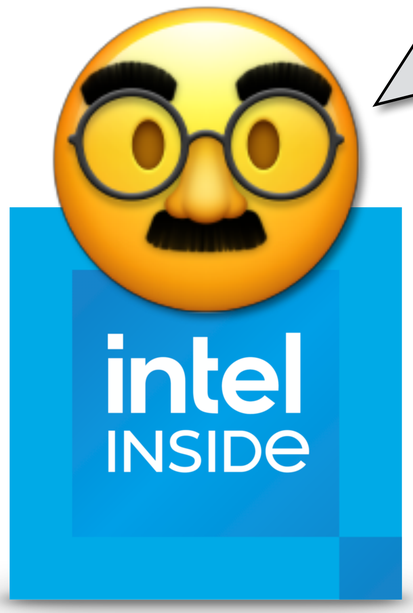
Sequential, scalar primitive computation

Uniform global heap memory

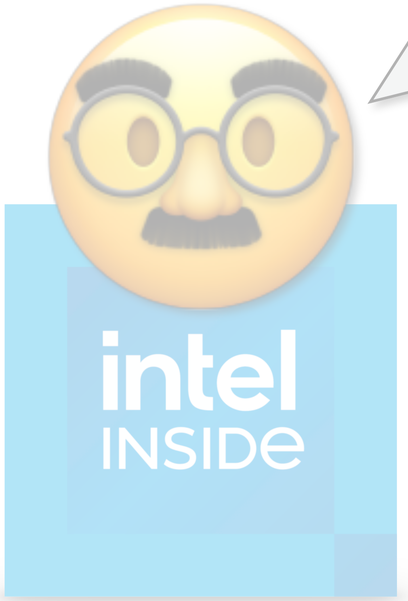
Abstraction through **subroutines & pointers**



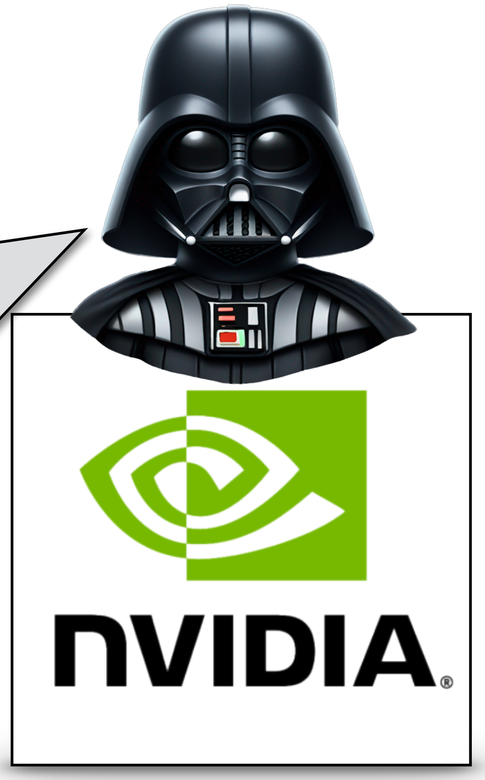
Why yes,
I am still
basically a
PDP-11



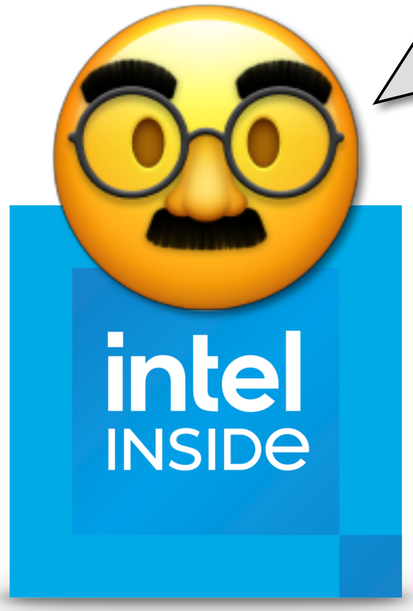
Why yes,
I am still
basically a
PDP-11



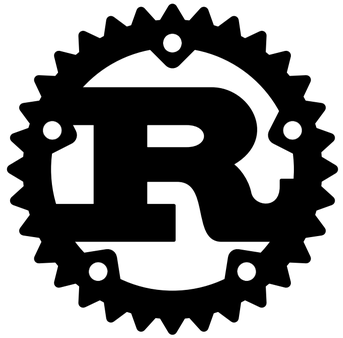
You will **rewrite**
all your code in
CUDA . . .
and pray I don't
alter the deal
any further!



Why yes,
I am still
basically a
PDP-11



Was there ever
anything else?



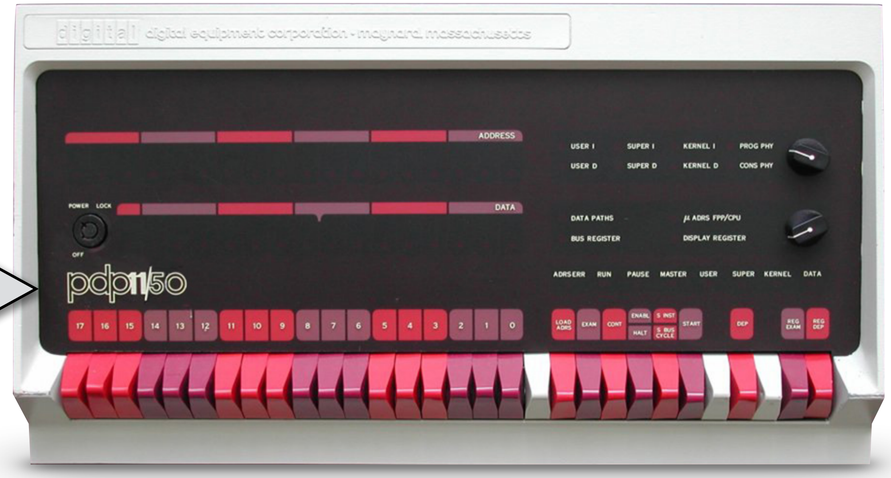
...

To **unlock** the **potential** of **hardware,**
we need to **rethink how we program.**

1. Better **understand hardware**

What makes hardware fast?

Executing a **sequential stream of instructions** in as **little time as possible**



A simple processor an **interpreter** for instructions!

1. Instr. fetch
2. Decode
3. Operand fetch
4. Execute
5. Write back

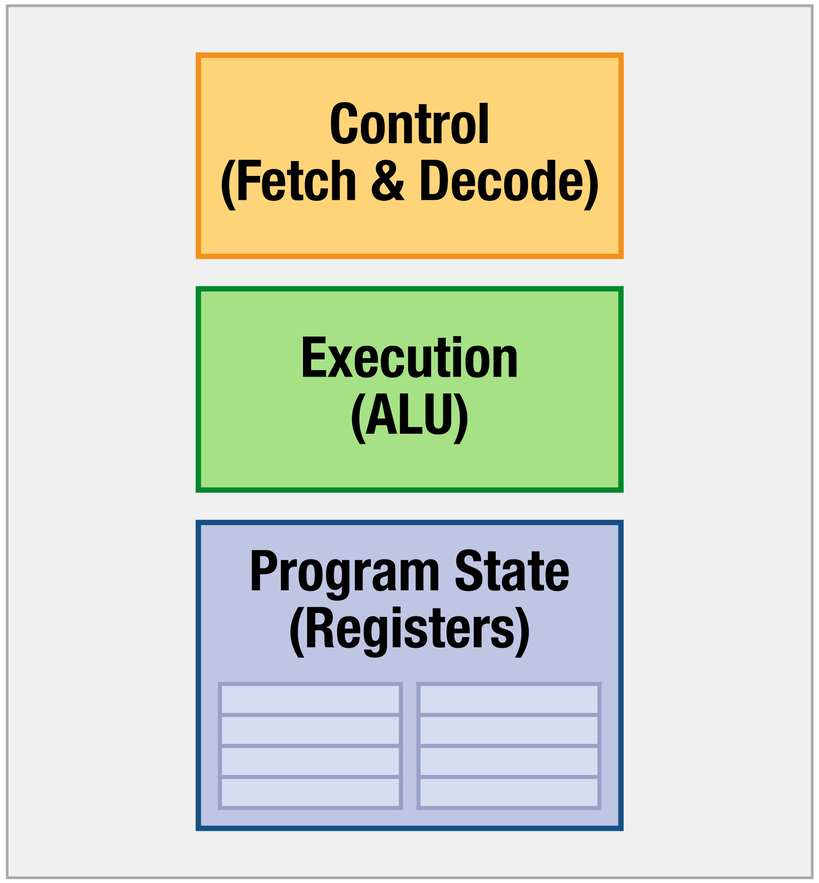
Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Program

Slides inspired by Kayvon Fatahalian

A simple processor an **interpreter** for instructions!



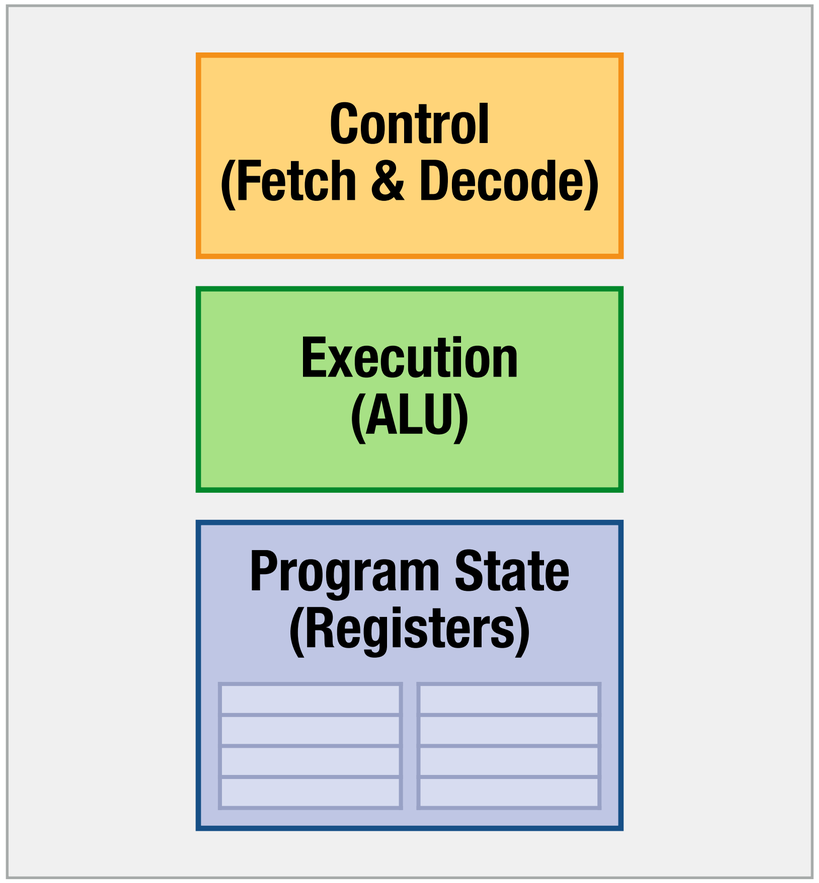
Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Program

Slides inspired by Kayvon Fatahalian

Optimization 1: Increase the **clock speed**



Processor

Higher **voltage**

↳ power grows with v^2

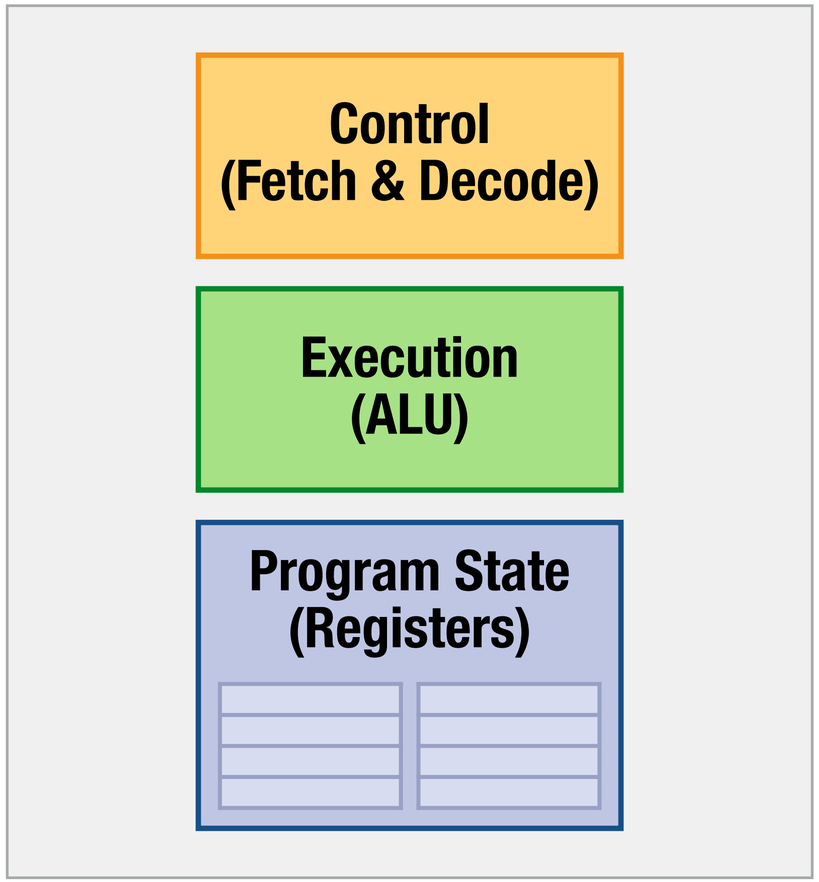
Faster **transistors**

↳ higher leakage

Deeper **pipelining**

↳ lower average IPC

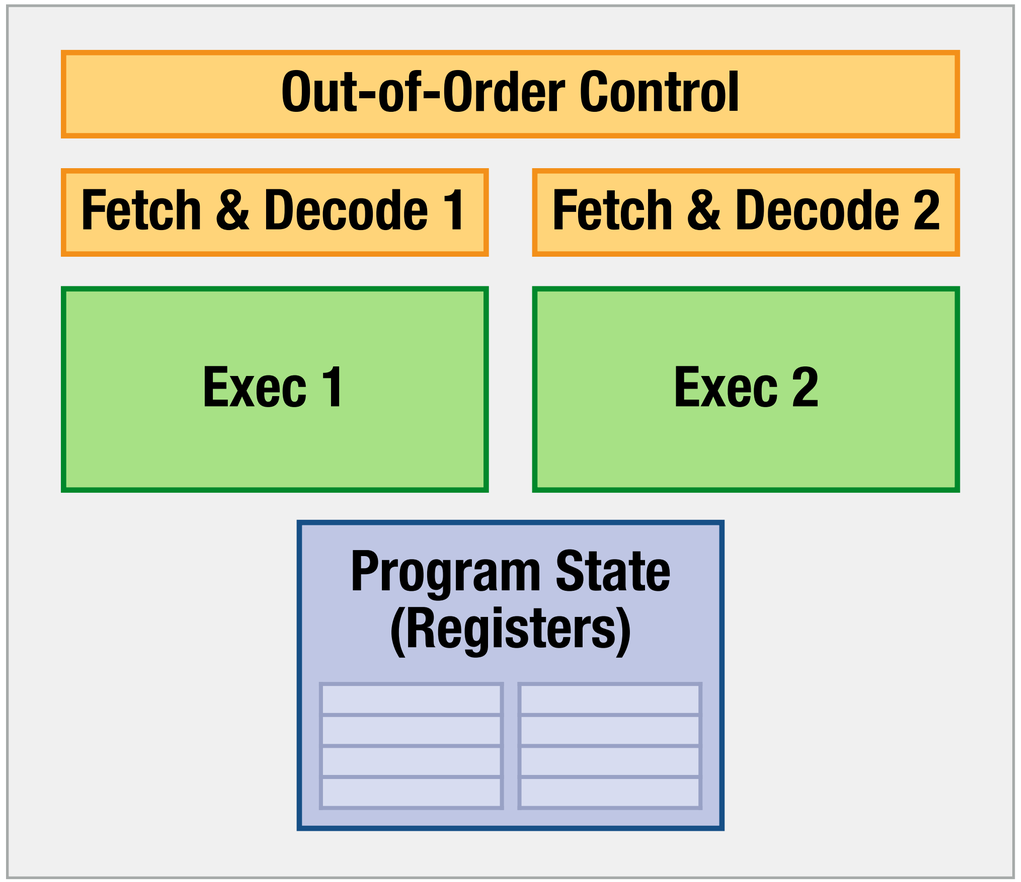
Optimization 2: Execute **multiple** instructions per cycle (**superscalar**)



Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

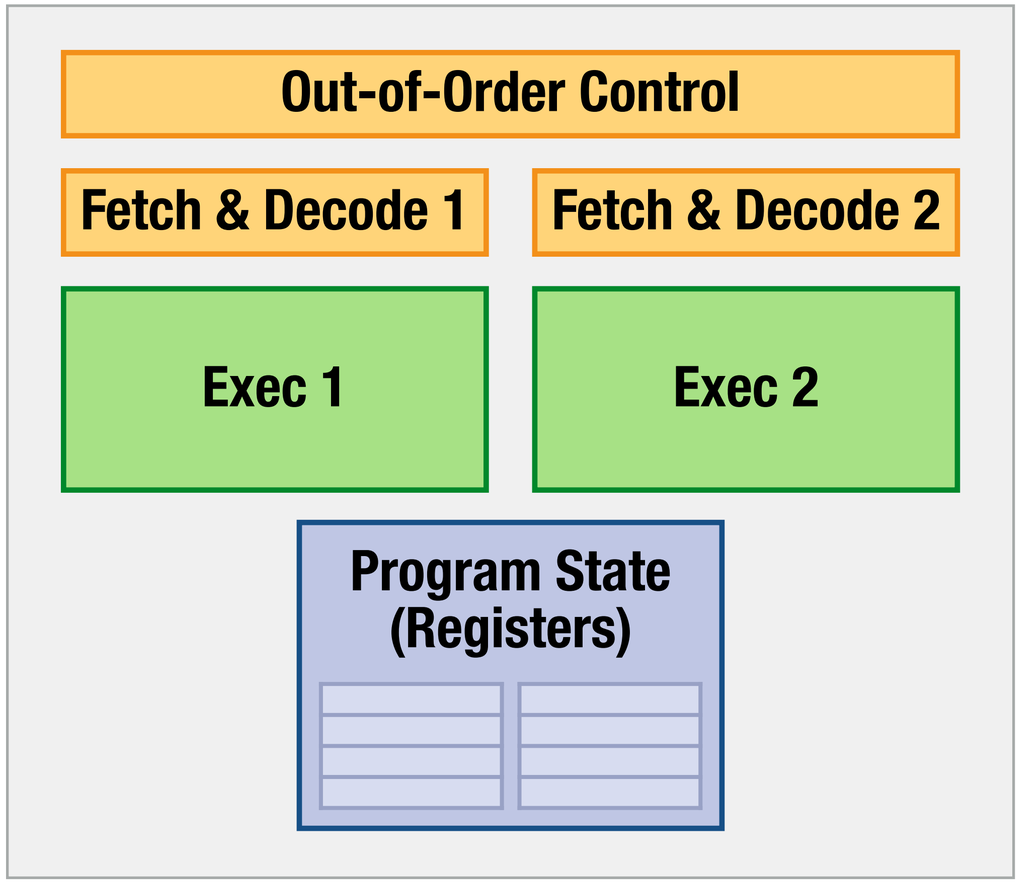
Optimization 2: Execute **multiple** instructions per cycle (**superscalar**)



Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

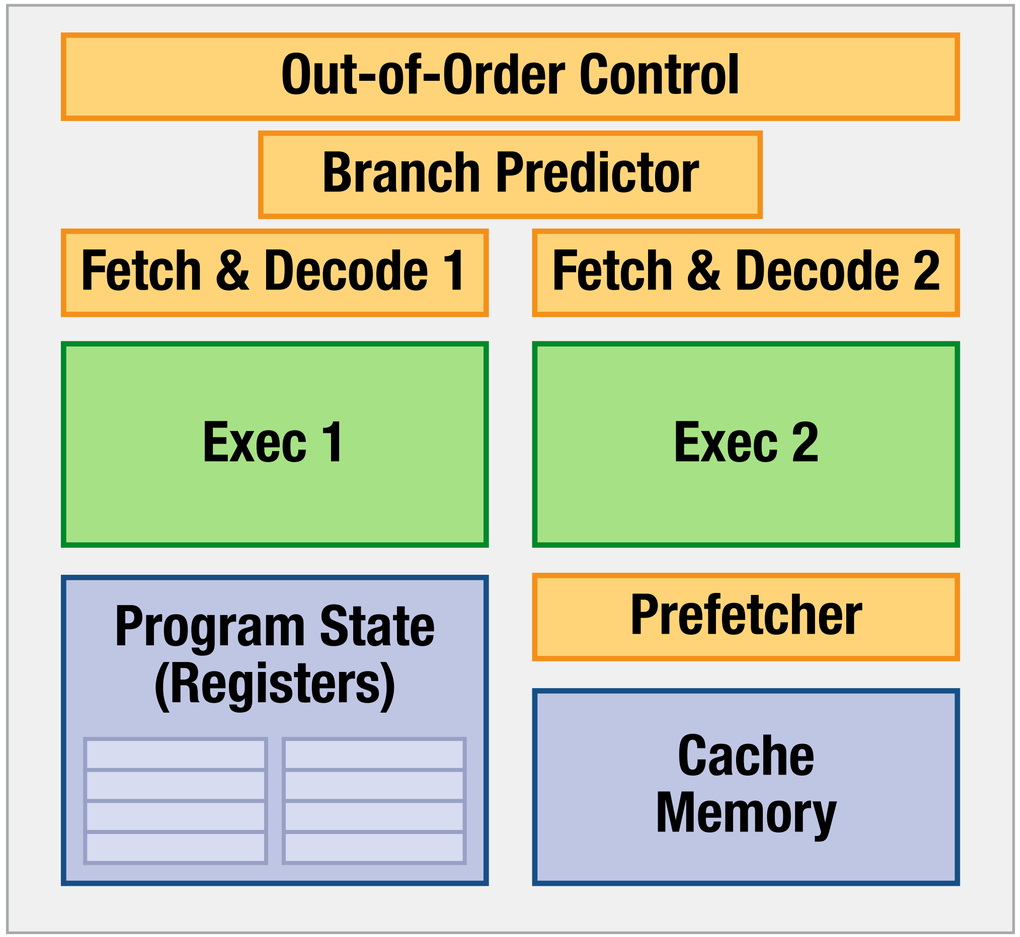
Optimization 3: Avoid **stalls** through **speculation**



Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

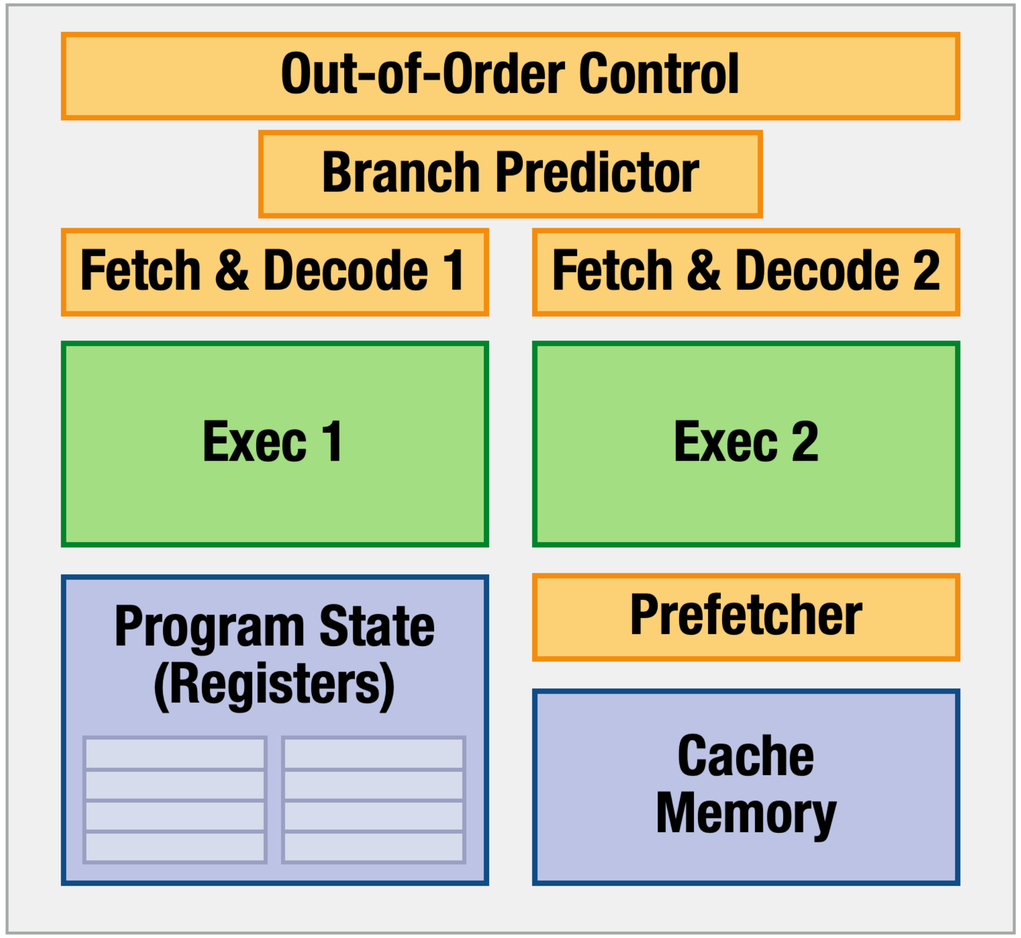
Optimization 3: Avoid **stalls** through **speculation**



Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

This makes sense when you have only **one processor**

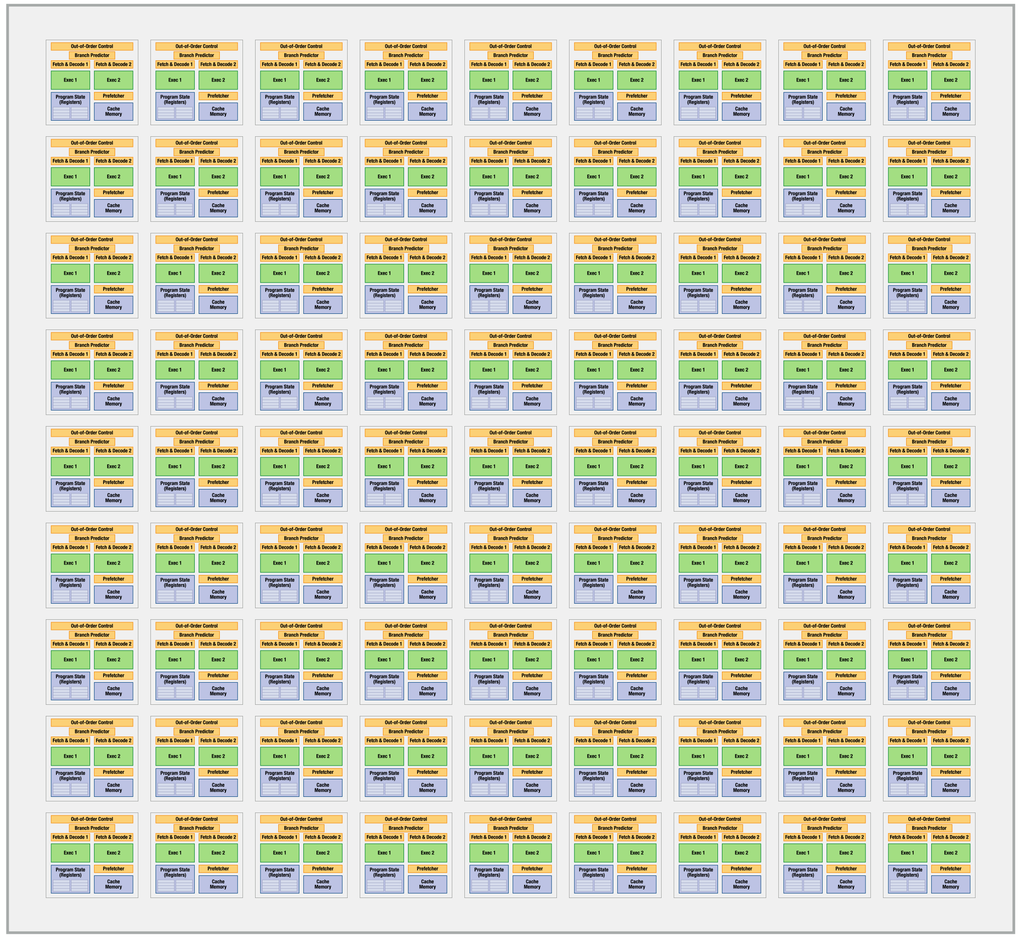


Processor

This makes sense when you have only **one processor**

Processor **(to scale)**

The reality today:



Silicon Chip

At the same time:

We **most need performance** when processing **lost of stuff**

↳ abundant **data-parallelism**

A **throughput-oriented** view of performance:

Optimize **aggregate rate** of processing **many items**

Three things drive throughput:

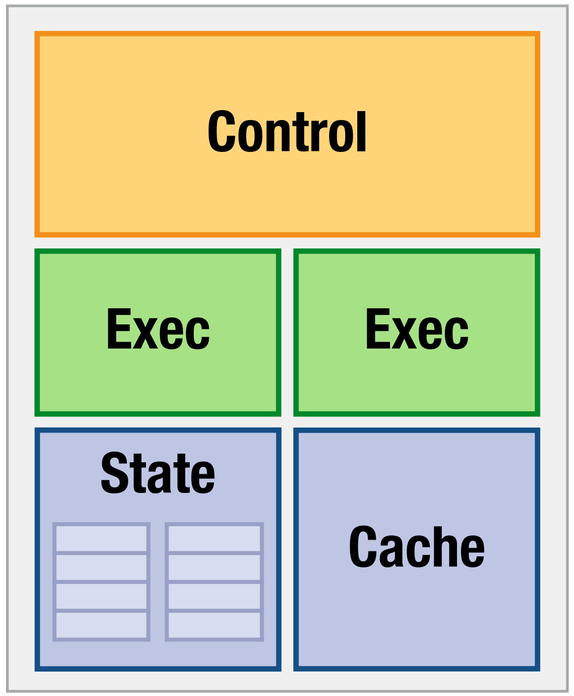
1 Amount of **work** to be done

2 Amount of **resources** to be applied (silicon, energy)

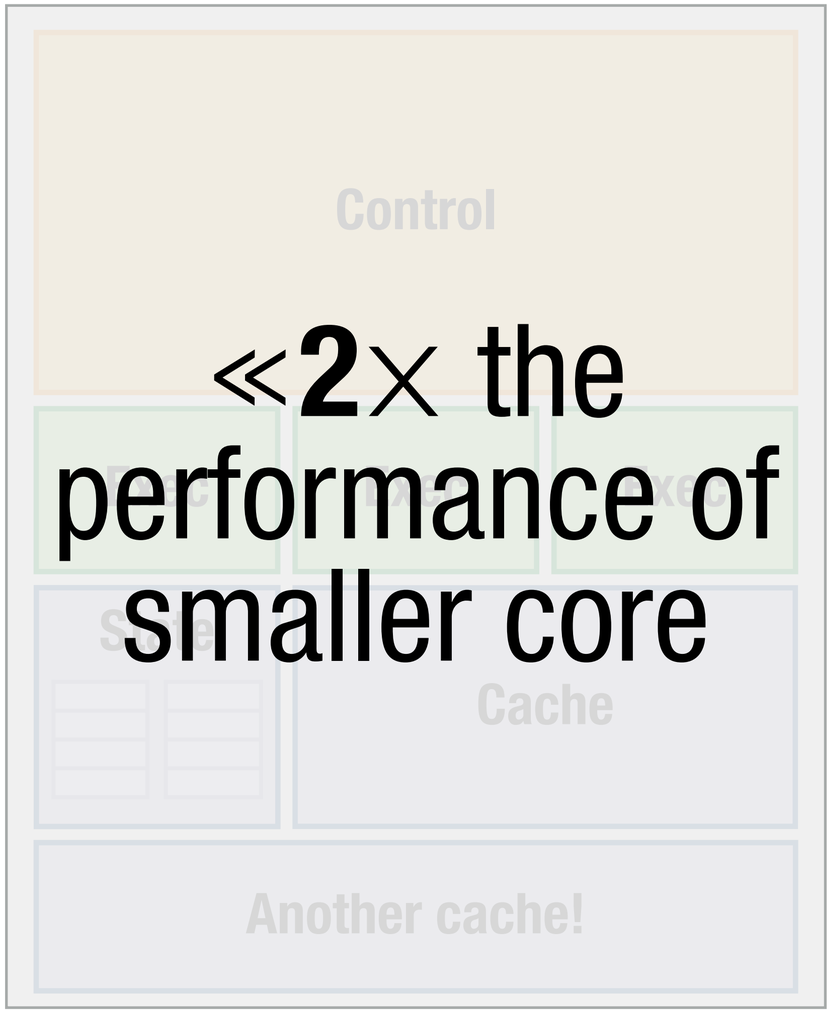
} **Constraints** set by application

3 **Efficiency** of applying them to useful work ← Goal: **optimize this!**

What to do with twice the silicon?



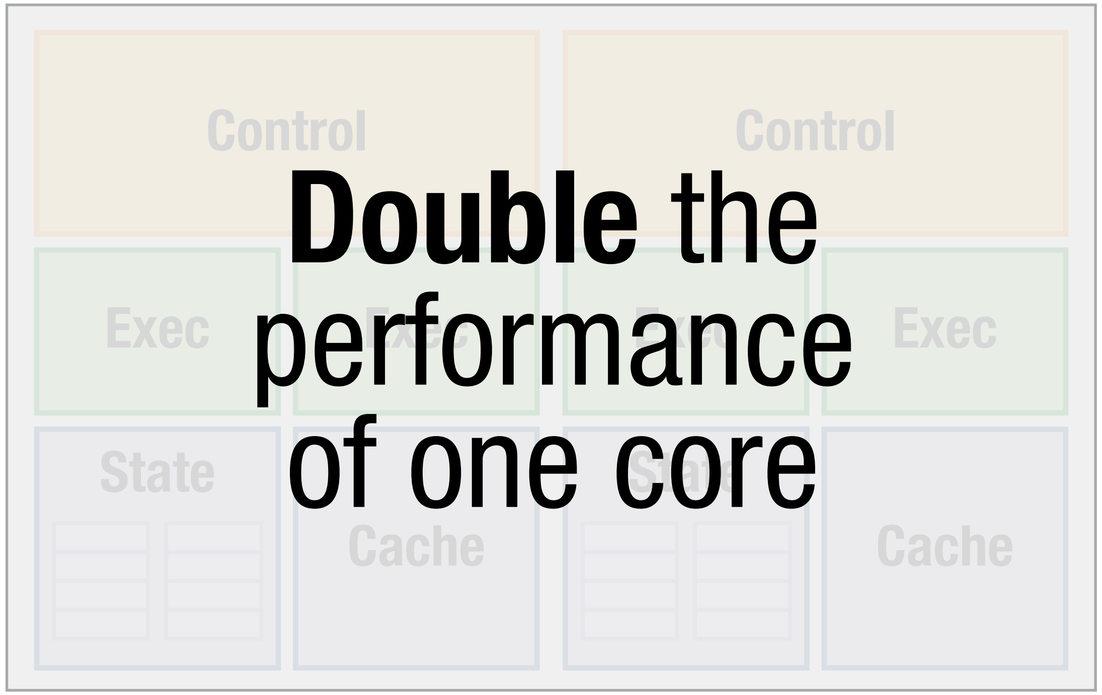
What to do with twice the silicon?



≪**2**× the performance of smaller core

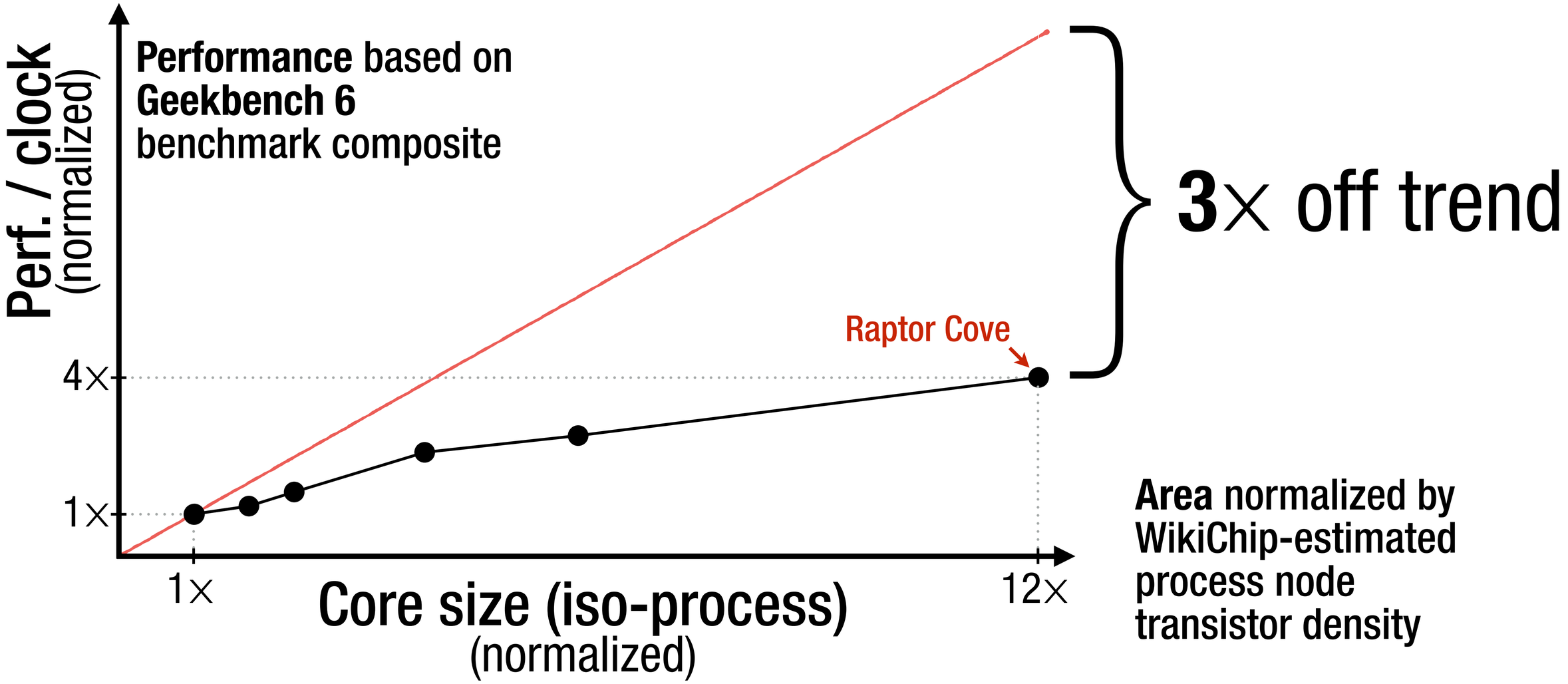
=

equal area



Double the performance of one core

Diminishing returns to **scaling single-core** performance



If we want to optimize **throughput,** is there a **better way** to **scale performance?**

Yes!

That’s the focus of this class.

1 Better **understand hardware**

2 **Practice programming** it for **high performance**

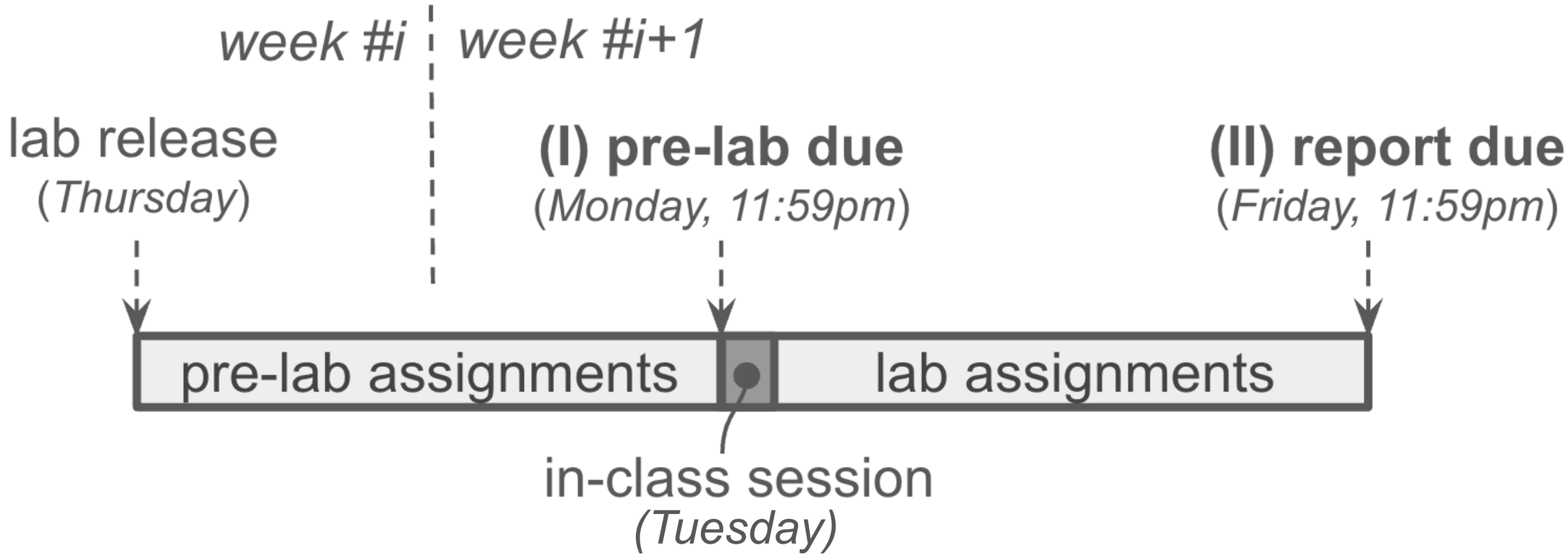
3 Reasoning about performance from **first principles**

1 **Lecture**
Thu 2:30-3:30pm

2 **Programming assignments**

3 **Lab session**
Tue 2:30-4:30pm

Weekly cadence:



First assignment out tonight,
first checkpoint due Monday (for Tuesday's lab)

Enrollment: 90 slots

Complete the survey today!

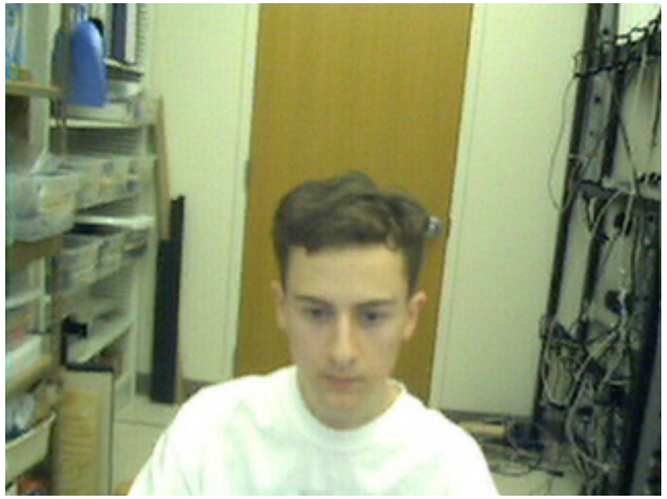
https://bit.ly/6s894-enrollment-2025


Expectations

Course web site:

accelerated-computing.academy

Your course staff



JRK

(Programming my first GPU, Stanford Graphics Lab, 2001)

Manya

Sobhan

Rishab

Questions?