Lecture 23: The GPU

Why It Exists, How It Is Built, and the Frame Budget

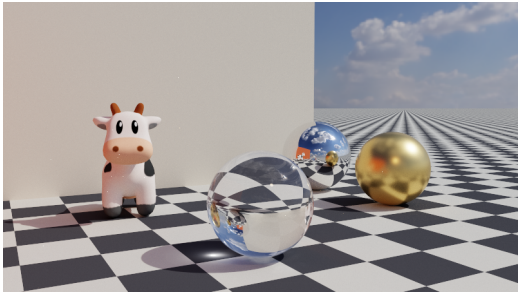
CS 453 — Computer Graphics

Fakhir Shaheen

Forman Christian University

Same scene: minutes against milliseconds

Two renderers written for this course, one scene

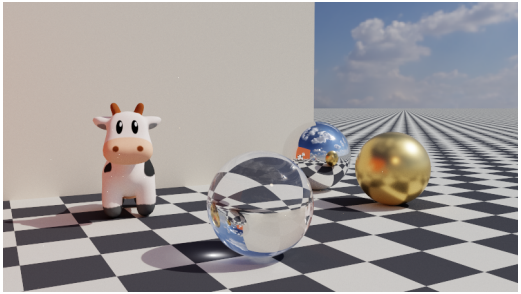


Part A, all out: our ray tracer, CPU

12 minutes on 12 cores

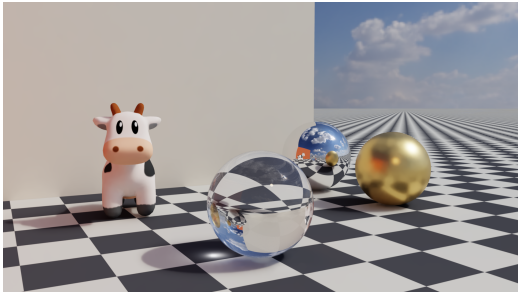
Same scene: minutes against milliseconds

Two renderers written for this course, one scene



Part A, all out: our ray tracer, CPU

12 minutes on 12 cores

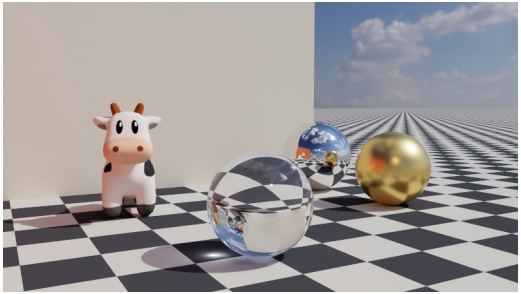


Real time: our renderer, GPU

13.5 ms a frame, RTX 5070

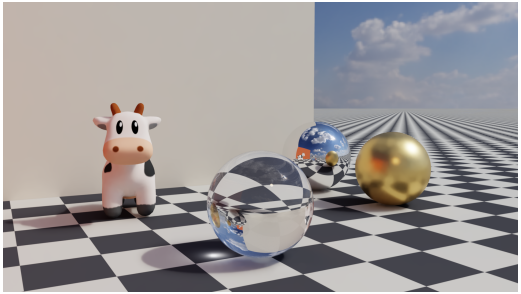
Same scene: minutes against milliseconds

Two renderers written for this course, one scene



Part A, all out: our ray tracer, CPU

12 minutes on 12 cores



Real time: our renderer, GPU

13.5 ms a frame, RTX 5070

Nearly the same picture: how is the right one so fast?

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s
GPU, the same brute force	800×450	$\approx 9{,}000$	1.88 s

$\div 390$ the machine

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s
GPU, the same brute force	800×450	$\approx 9{,}000$	1.88 s
GPU, the same brute force	1920×1080	$\approx 9{,}000$	8.61 s

$\div 390$ the machine

$\times 4.6$ more pixels

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s
GPU, the same brute force	800×450	$\approx 9{,}000$	1.88 s
GPU, the same brute force	1920×1080	$\approx 9{,}000$	8.61 s
GPU, real time (slide 2, right)	1920×1080	≈ 9	0.0135 s

$\div 390$ the machine

$\times 4.6$ more pixels

$\div 640$ the tricks

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s
GPU, the same brute force	800×450	$\approx 9{,}000$	1.88 s
GPU, the same brute force	1920×1080	$\approx 9{,}000$	8.61 s
GPU, real time (slide 2, right)	1920×1080	≈ 9	0.0135 s

$\div 390$ the machine

$\times 4.6$ more pixels

$\div 640$ the tricks

- The machine alone gives a factor of 390: that is today

Same picture, 55,000 times faster: where it goes

Run the slow algorithm on the GPU too, and the factor splits in two

who renders	picture	rays per pixel	time
Python, 12 CPU cores (slide 2, left)	800×450	$\approx 9{,}000$	737 s
GPU, the same brute force	800×450	$\approx 9{,}000$	1.88 s
GPU, the same brute force	1920×1080	$\approx 9{,}000$	8.61 s
GPU, real time (slide 2, right)	1920×1080	≈ 9	0.0135 s

$\div 390$ the machine

$\times 4.6$ more pixels

$\div 640$ the tricks

- The machine alone gives a factor of 390: that is today
- The tricks give the other 640: far fewer rays, the rest borrowed

Today: four questions about the machine

Where the factor of 390 comes from, and what the machine cannot afford

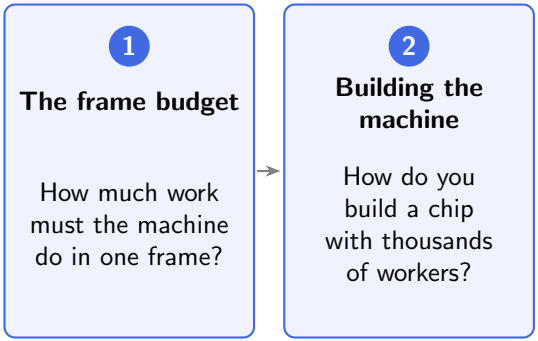
1

The frame budget

How much work must the machine do in one frame?

Today: four questions about the machine

Where the factor of 390 comes from, and what the machine cannot afford



Today: four questions about the machine

Where the factor of 390 comes from, and what the machine cannot afford

1 **The frame budget**

How much work must the machine do in one frame?

→

2 **Building the machine**

How do you build a chip with thousands of workers?

→

3 **A real GPU**

Does a real GPU look like our design?

Today: four questions about the machine

Where the factor of 390 comes from, and what the machine cannot afford

1 **The frame budget**

How much work must the machine do in one frame?

→

2 **Building the machine**

How do you build a chip with thousands of workers?

→

3 **A real GPU**

Does a real GPU look like our design?

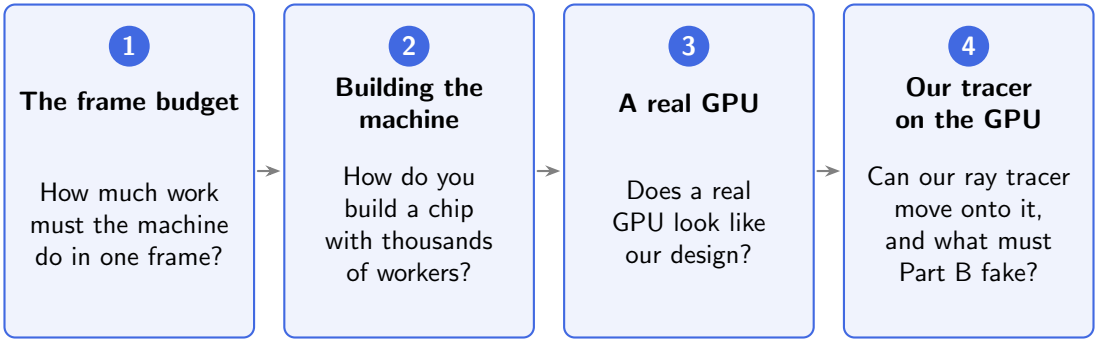
→

4 **Our tracer on the GPU**

Can our ray tracer move onto it, and what must Part B fake?

Today: four questions about the machine

Where the factor of 390 comes from, and what the machine cannot afford

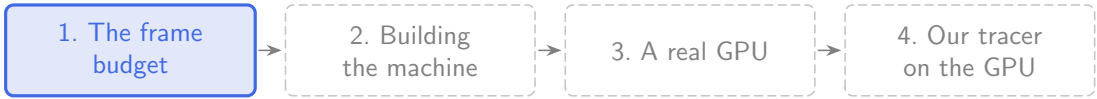


- Each part ends with the question that opens the next one

  The tricks, the other factor of 640, fill the rest of Part B, from lecture 24 on

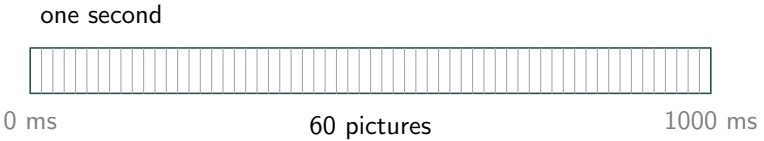
1. The frame budget

How much work must the machine do in one frame?



A new picture every 16.7 ms

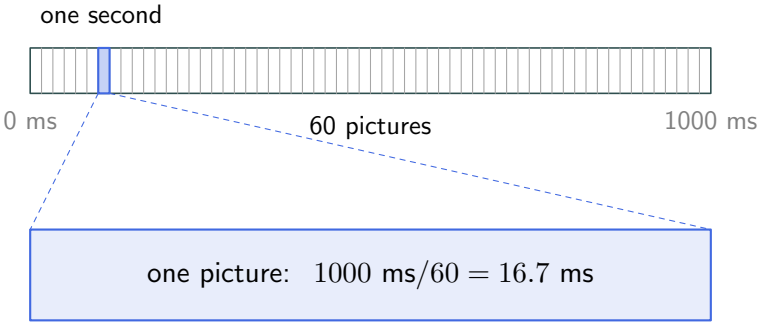
First the deadline: how long does one picture get?



- Smooth motion: 60 pictures a second

A new picture every 16.7 ms

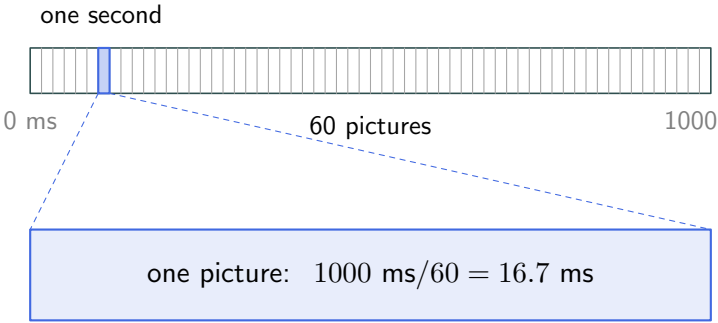
First the deadline: how long does one picture get?



- Smooth motion: 60 pictures a second
- Each picture gets 16.7 ms

A new picture every 16.7 ms

First the deadline: how long does one picture get?

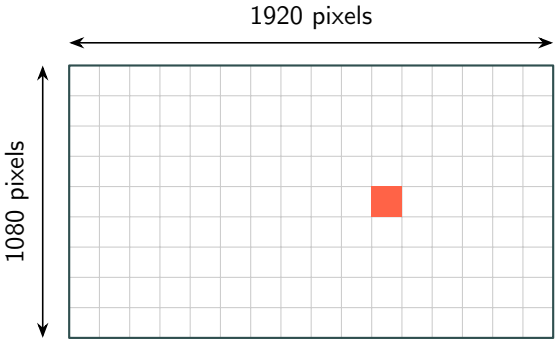


- Smooth motion: 60 pictures a second
- Each picture gets 16.7 ms
- A late picture: the motion stutters

other screens: 30 a second: 33.3 ms 90 a second (VR): 11.1 ms
120 a second: 8.3 ms

Two million pixels share each frame

How many pixels share those 16.7 ms?

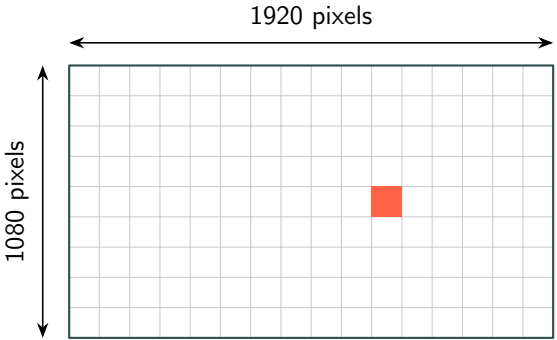


(drawn coarse: each square stands for many pixels)

$$1920 \times 1080 = 2{,}073{,}600 \text{ pixels}$$

Two million pixels share each frame

How many pixels share those 16.7 ms?



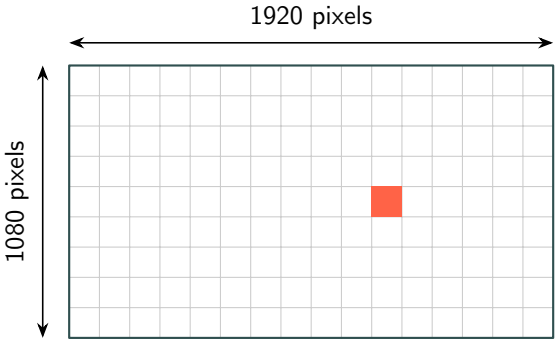
(drawn coarse: each square stands for many pixels)

$1920 \times 1080 = 2{,}073{,}600$ pixels

$\times\ 60 = 124{,}416{,}000$ pixels a second

Two million pixels share each frame

How many pixels share those 16.7 ms?



(drawn coarse: each square stands for many pixels)

$1920 \times 1080 = 2{,}073{,}600$ pixels

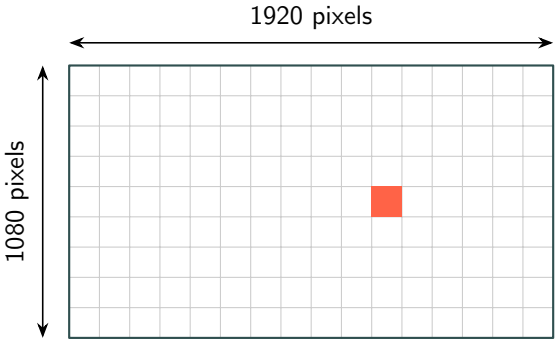
$\times\ 60 = 124{,}416{,}000$ pixels a second

$$\frac{16.7 \text{ ms}}{2{,}073{,}600} = 8.0 \text{ ns per pixel}$$

ns: a nanosecond, a billionth of a second

Two million pixels share each frame

How many pixels share those 16.7 ms?



(drawn coarse: each square stands for many pixels)

$1920 \times 1080 = 2{,}073{,}600$ pixels

$\times\ 60 = 124{,}416{,}000$ pixels a second

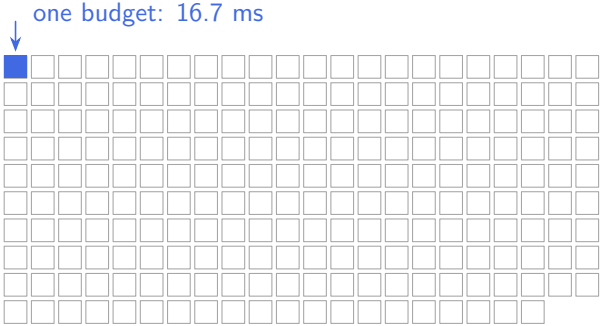
$$\frac{16.7 \text{ ms}}{2{,}073{,}600} = 8.0 \text{ ns per pixel}$$

ns: a nanosecond, a billionth of a second

- Each pixel gets about 8 ns for all of its work

Our tracer: 3.6 seconds for one frame

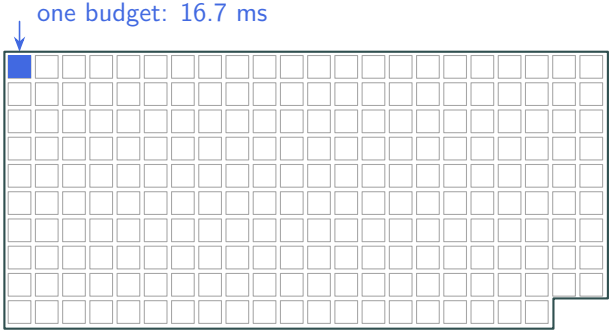
How far is our Part A tracer from that budget?



- We timed lecture 19's room at 1920×1080

Our tracer: 3.6 seconds for one frame

How far is our Part A tracer from that budget?

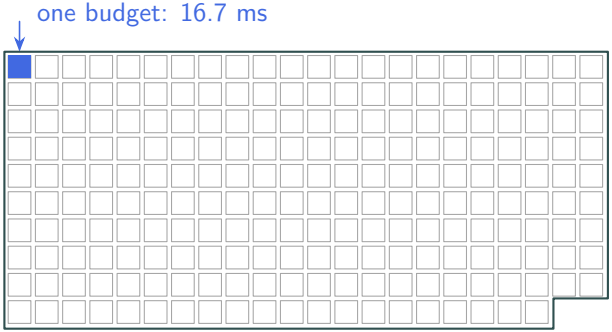


- We timed lecture 19's room at 1920×1080
- Python on one core is 218 times too slow

Whitted, 1 ray a pixel: **3.6 s** = 218 budgets

Our tracer: 3.6 seconds for one frame

How far is our Part A tracer from that budget?



- We timed lecture 19's room at 1920×1080
- Python on one core is 218 times too slow
- Bounced light costs far more again

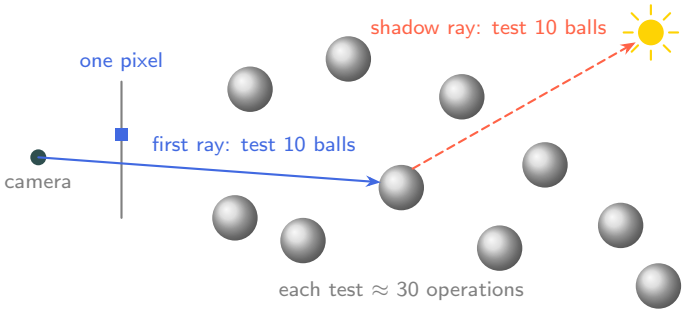
Whitted, 1 ray a pixel: **3.6 s** = 218 budgets

12 bounces (lecture 20), 1 ray: **47 s** = 2,800 budgets

256 rays a pixel: 3.3 hours

Even perfect code needs 15 times the time

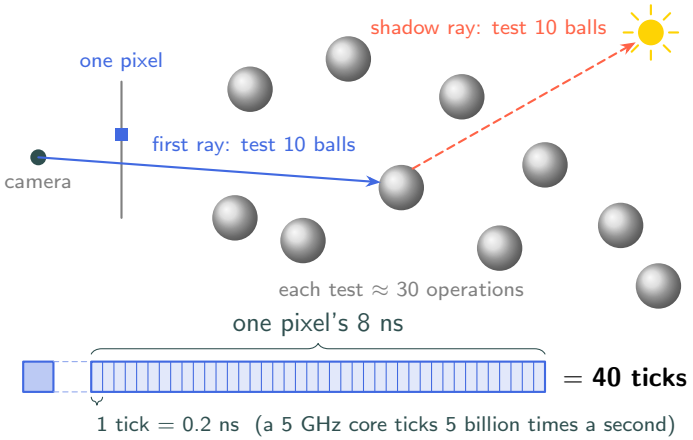
Is the 218 just Python's fault? Suppose the code were perfect



- One pixel, a toy scene: two rays, ten balls

Even perfect code needs 15 times the time

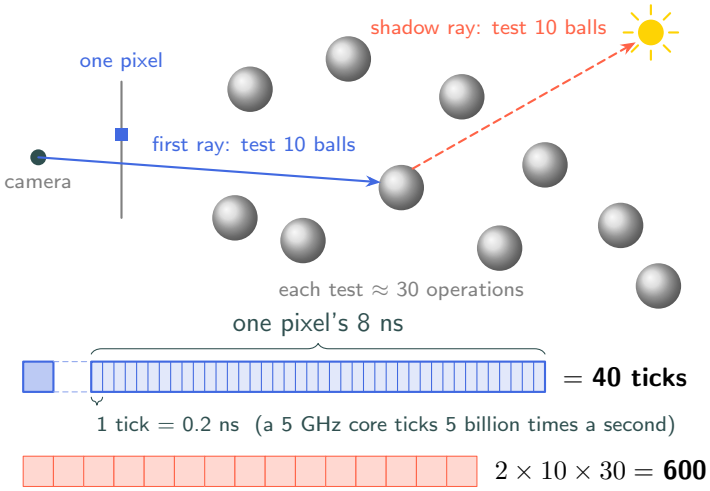
Is the 218 just Python's fault? Suppose the code were perfect



- One pixel, a toy scene: two rays, ten balls
- Its 8 ns: room for 40 operations

Even perfect code needs 15 times the time

Is the 218 just Python's fault? Suppose the code were perfect



- One pixel, a toy scene: two rays, ten balls
- Its 8 ns: room for 40 operations
- **It needs 600: 15 times too many, with no Python at all**

A cleverer core helps, but not enough

What if each tick did more than one operation?

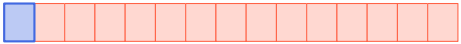
1 operation a tick



A cleverer core helps, but not enough

What if each tick did more than one operation?

1 operation a tick = one pixel's time (slide 9)



15×

4 operations a tick (a modern core does several at once)

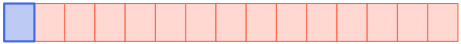


still almost 4×

A cleverer core helps, but not enough

What if each tick did more than one operation?

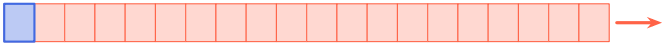
1 operation a tick = one pixel's time (slide 9)



4 operations a tick (a modern core does several at once)



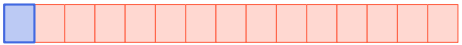
a real frame: more lamps, more rays, thousands of triangles



A cleverer core helps, but not enough

What if each tick did more than one operation?

1 operation a tick = one pixel's time (slide 9)

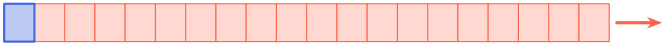


4 operations a tick (a modern core does several at once)



still almost 4×

a real frame: more lamps, more rays, thousands of triangles





one core: 2 million pixels wait in line



many cores: many pixels at once

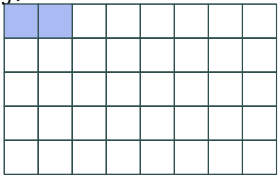
The way out is not faster code: it is many workers at once

Every pixel runs the same small program

Can many workers share one frame? Only if no pixel waits for another

```
for j in range(height):
    for i in range(width):
        image[j][i] = shade(i, j)
```

Part A: one pixel after another



pixels

- Every pixel runs the same steps on its own numbers

Every pixel runs the same small program

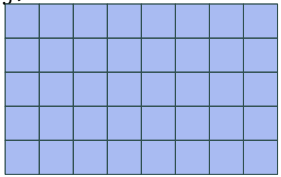
Can many workers share one frame? Only if no pixel waits for another

```
for j in range(height):
    for i in range(width):
        image[j][i] = shade(i, j)
```

Part A: one pixel after another

`shade(i, j)`

same steps, all pixels at once



pixels

- Every pixel runs the same steps on its own numbers
- No pixel waits for another pixel's answer

Every pixel runs the same small program

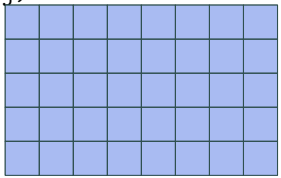
Can many workers share one frame? Only if no pixel waits for another

```
for j in range(height):
    for i in range(width):
        image[j][i] = shade(i, j)
```

Part A: one pixel after another

`shade(i, j)`

same steps, all pixels at once

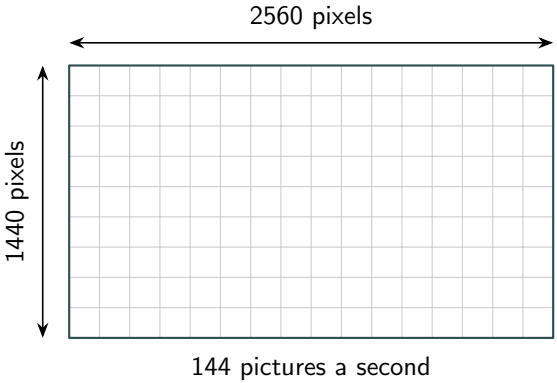


pixels

- Every pixel runs the same steps on its own numbers
- No pixel waits for another pixel's answer
- So thousands of workers can share a frame

Quick check 1: the budget at 144 Hz

Check: the budget on a gaming monitor



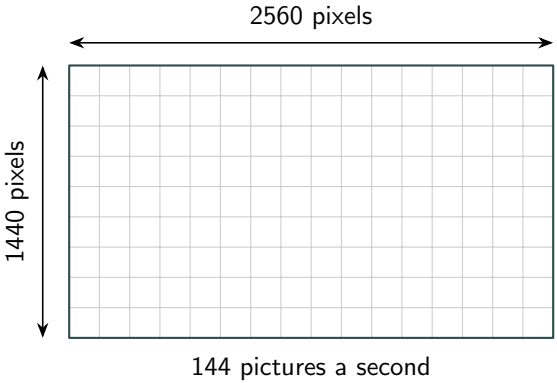
(a) milliseconds per picture?

(b) pixels per picture?

(c) nanoseconds per pixel?

Quick check 1: the budget at 144 Hz

Check: the budget on a gaming monitor



(a) milliseconds per picture?

$1000/144 = 6.94$ ms

(b) pixels per picture?

$2560 \times 1440 = 3{,}686{,}400$

(c) nanoseconds per pixel?

6.94 ms$/3{,}686{,}400 = 1.9$ ns

- A quarter of full HD's 8 ns

So far: 8 ns a pixel, and no pixel waits

$$\text{time per pixel} = \frac{1000 \text{ ms} / \text{pictures a second}}{\text{pixels in a picture}}$$

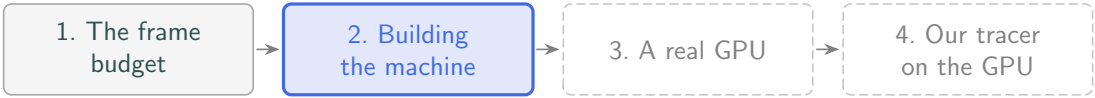
$$\frac{16.7 \text{ ms}}{2{,}073{,}600} \approx 8 \text{ ns:} \quad \text{one core's 40 ticks, for about 600 operations}$$

Every pixel's work is independent, so many workers can share a frame

Next question: how do you build a chip with thousands of workers?

2. Building the machine

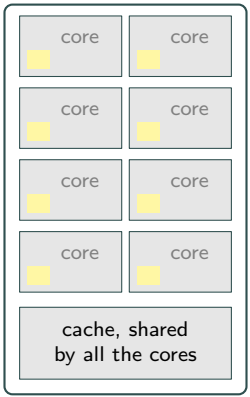
How do you build a chip with thousands of workers?



A CPU core is built to finish one job fast

Start from the chip that runs our Python today, and zoom into one core

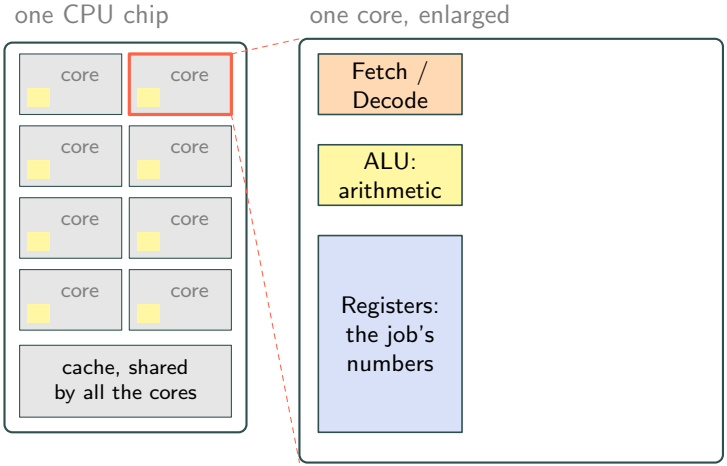
one CPU chip



- A CPU chip holds a few cores and a cache

A CPU core is built to finish one job fast

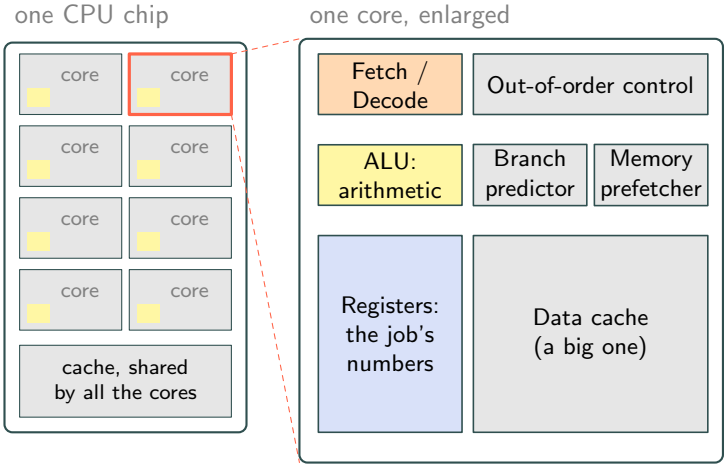
Start from the chip that runs our Python today, and zoom into one core



- A CPU chip holds a few cores and a cache
- In each core, only the yellow ALU does arithmetic

A CPU core is built to finish one job fast

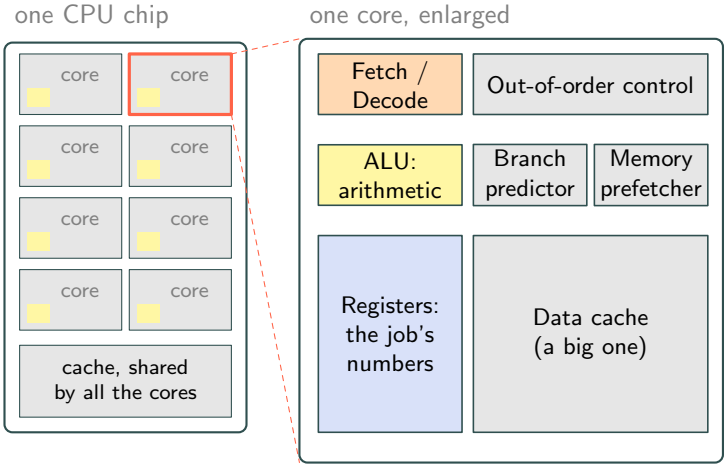
Start from the chip that runs our Python today, and zoom into one core



- A CPU chip holds a few cores and a cache
- In each core, only the yellow ALU does arithmetic
- The grey parts make one job finish sooner

A CPU core is built to finish one job fast

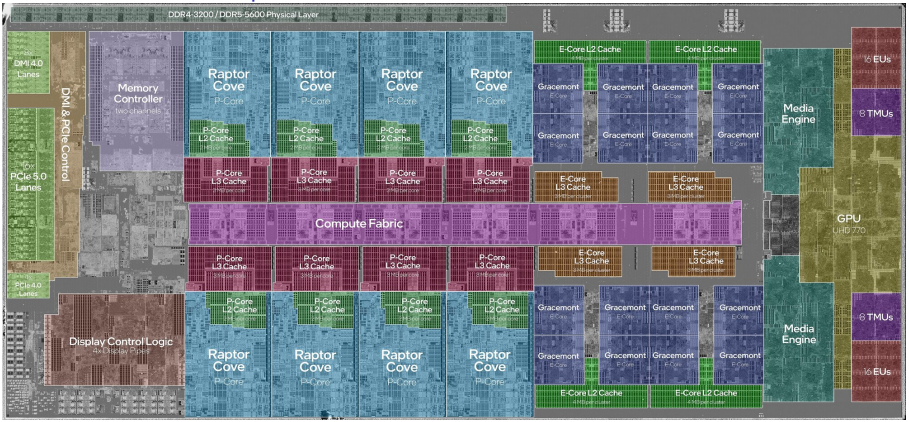
Start from the chip that runs our Python today, and zoom into one core



- A CPU chip holds a few cores and a cache
- In each core, only the yellow ALU does arithmetic
- The grey parts make one job finish sooner
- They guess, reorder and remember, but compute nothing

A real CPU: a few big cores, much cache

On a real CPU, how much of the chip does arithmetic?

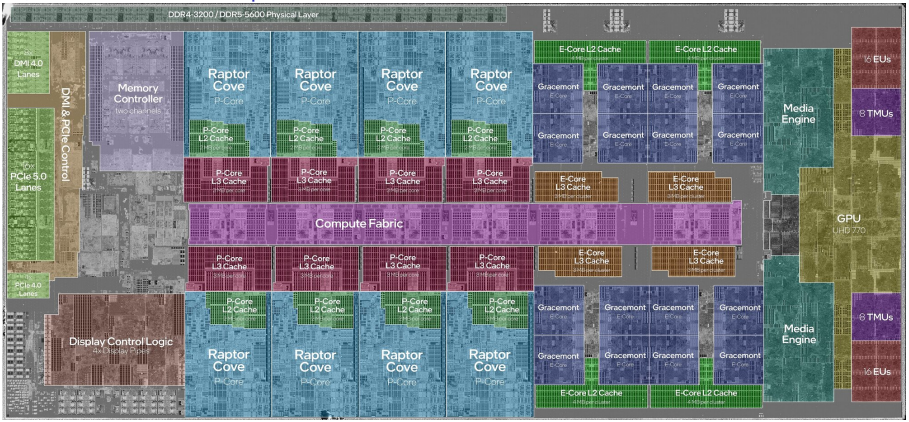


- Blue: 8 big cores.
  Purple: 16 small cores.

Intel Core i9-13900K; labels by JmsDoug, CC0; see images/CREDITS.txt

A real CPU: a few big cores, much cache

On a real CPU, how much of the chip does arithmetic?

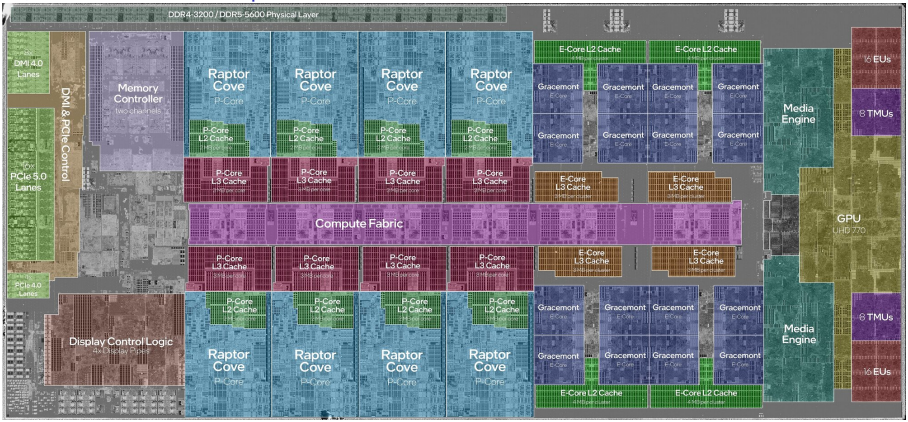


- Blue: 8 big cores.
  Purple: 16 small cores.
- Red, green, orange: caches

Intel Core i9-13900K; labels by JmsDoug, CC0; see images/CREDITS.txt

A real CPU: a few big cores, much cache

On a real CPU, how much of the chip does arithmetic?

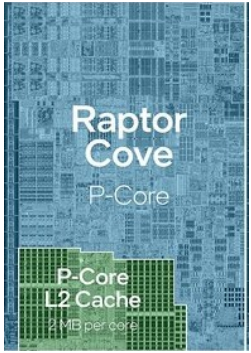


- Blue: 8 big cores. Purple: 16 small cores.
- Red, green, orange: caches
- Arithmetic: inside each core, too small to see

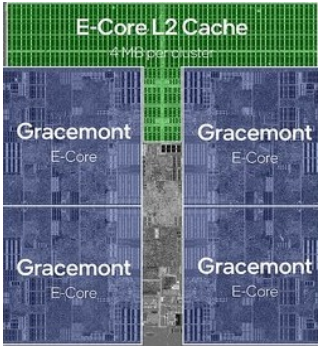
Intel Core i9-13900K; labels by JmsDoug, CC0; see images/CREDITS.txt

Big cores or small cores, in the same area

The CPU on the last slide already makes this trade



1 big core



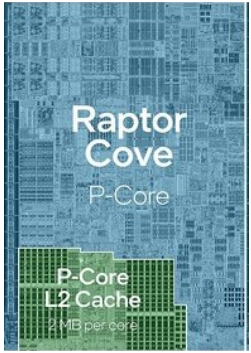
4 small cores

- Four small cores fit in about the area of one big core

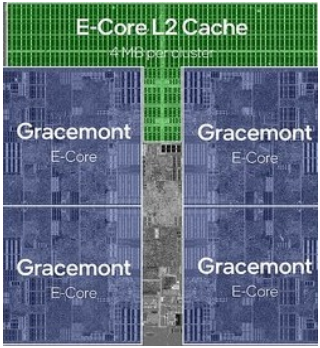
Both cut from slide 16's photo at the same scale, each with its L2 cache

Big cores or small cores, in the same area

The CPU on the last slide already makes this trade



1 big core



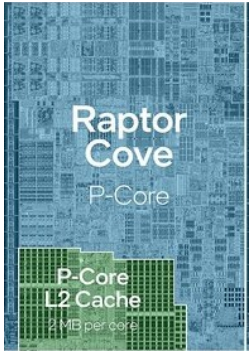
4 small cores

Both cut from slide 16's photo at the same scale, each with its L2 cache

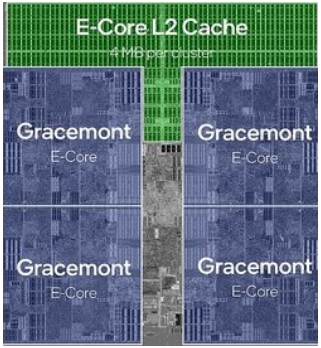
- Four small cores fit in about the area of one big core
- A small core has fewer helpers: one job runs slower on it

Big cores or small cores, in the same area

The CPU on the last slide already makes this trade



1 big core



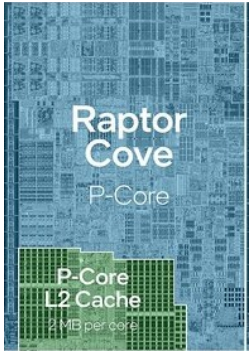
4 small cores

Both cut from slide 16's photo at the same scale, each with its L2 cache

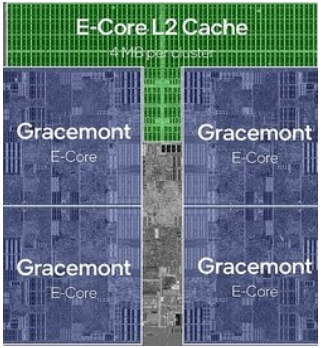
- Four small cores fit in about the area of one big core
- A small core has fewer helpers: one job runs slower on it
- Many jobs at once: the four small cores get more done

Big cores or small cores, in the same area

The CPU on the last slide already makes this trade



1 big core



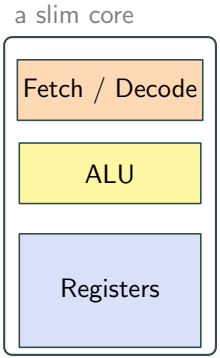
4 small cores

Both cut from slide 16's photo at the same scale, each with its L2 cache

- Four small cores fit in about the area of one big core
- A small core has fewer helpers: one job runs slower on it
- Many jobs at once: the four small cores get more done
- A GPU takes this trade much further

Idea 1: strip the core down, then copy it

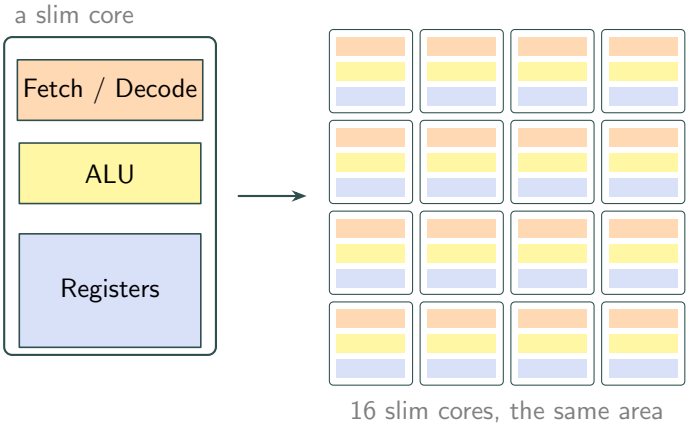
We want two million pixels done, not one pixel done fast



- Drop the helpers that speed up one job

Idea 1: strip the core down, then copy it

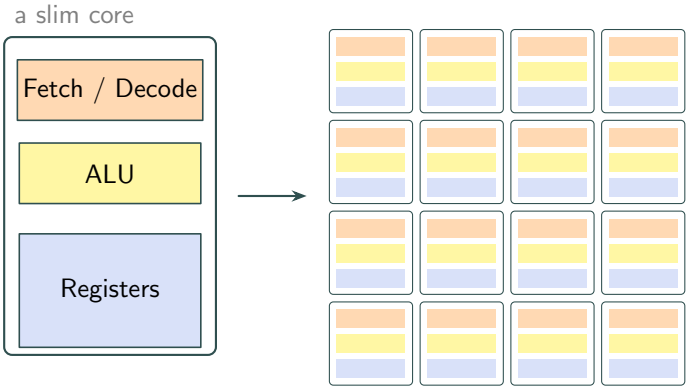
We want two million pixels done, not one pixel done fast



- Drop the helpers that speed up one job
- Spend their area on more slim cores

Idea 1: strip the core down, then copy it

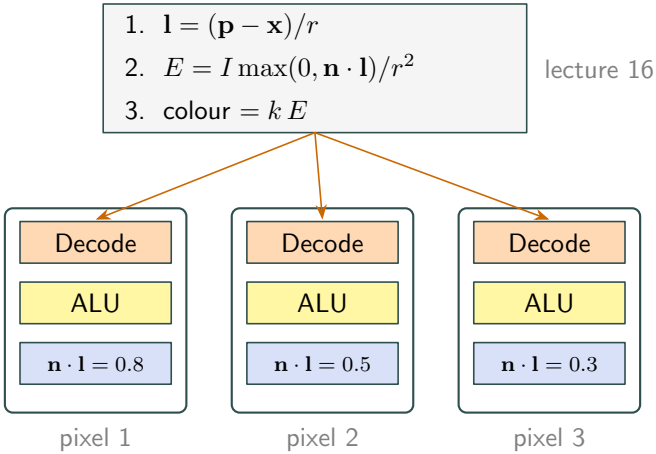
We want two million pixels done, not one pixel done fast



- Drop the helpers that speed up one job
- Spend their area on more slim cores
- 16 cores shade 16 pixels at once

Neighbouring pixels run the very same steps

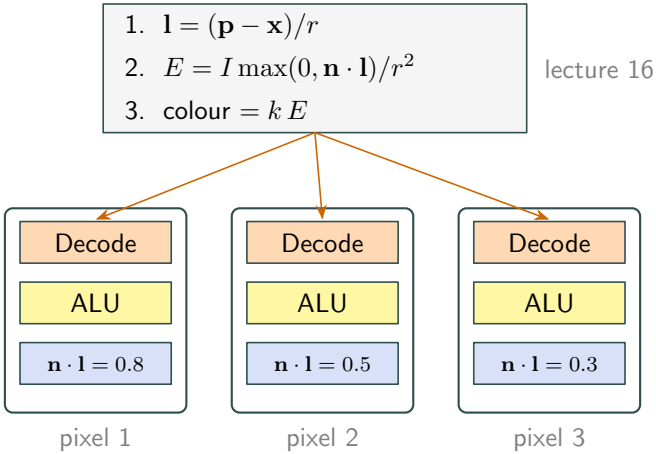
Look at what those 16 cores are doing



- Every core runs one list of steps

Neighbouring pixels run the very same steps

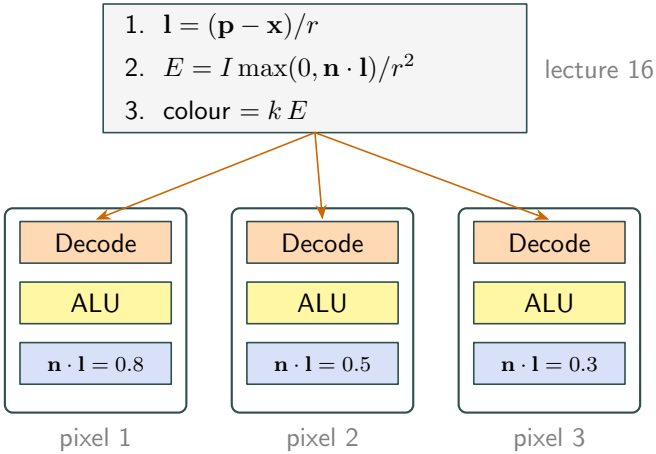
Look at what those 16 cores are doing



- Every core runs one list of steps
- Only the numbers differ

Neighbouring pixels run the very same steps

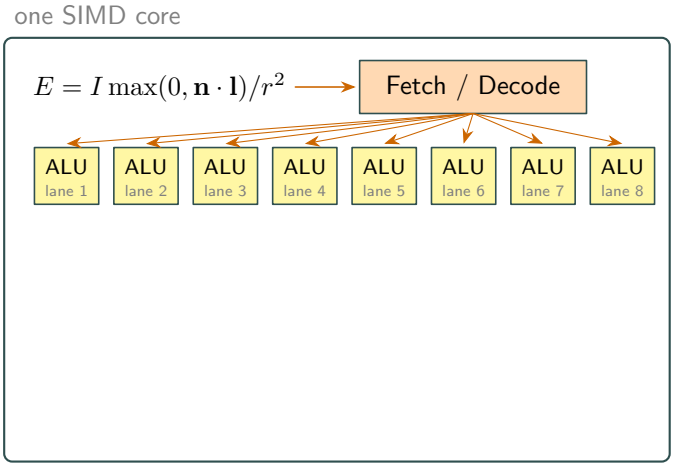
Look at what those 16 cores are doing



- Every core runs one list of steps
- Only the numbers differ
- So why decode each step 16 times?

Idea 2: one decoder drives eight lanes

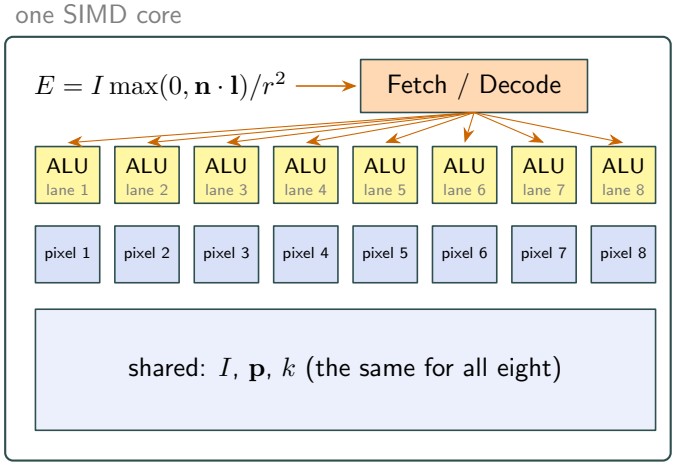
Remove that waste: decode once, compute many times



- One decoder reads each step once

Idea 2: one decoder drives eight lanes

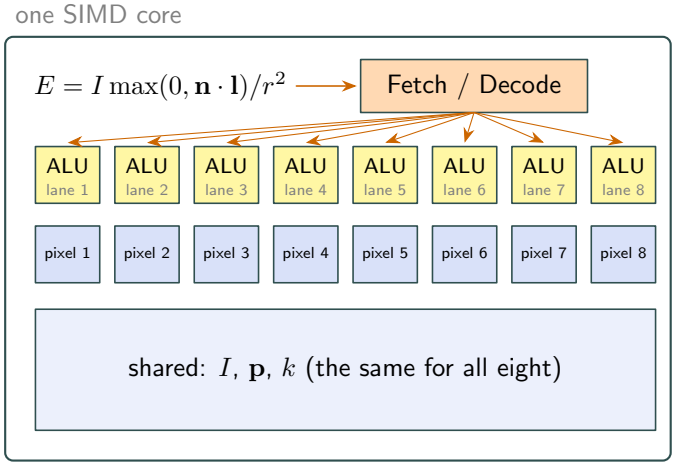
Remove that waste: decode once, compute many times



- One decoder reads each step once
- Each ALU, a *lane*, runs one pixel's copy: a *thread*

Idea 2: one decoder drives eight lanes

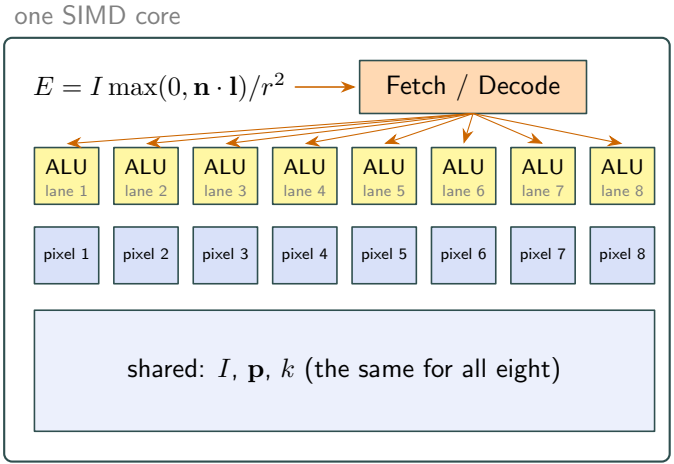
Remove that waste: decode once, compute many times



- One decoder reads each step once
- Each ALU, a *lane*, runs one pixel's copy: a *thread*
- SIMD: single instruction, multiple data

Idea 2: one decoder drives eight lanes

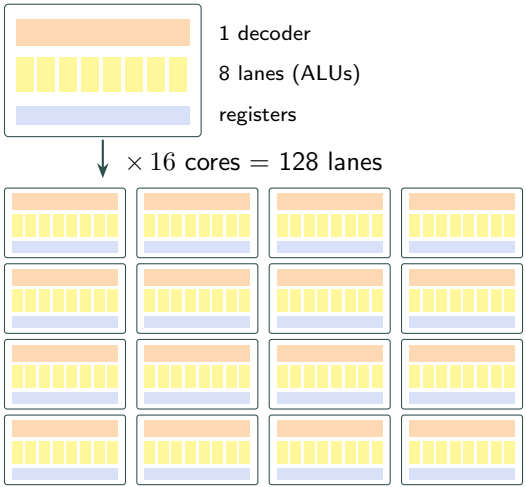
Remove that waste: decode once, compute many times



- One decoder reads each step once
- Each ALU, a *lane*, runs one pixel's copy: a *thread*
- SIMD: single instruction, multiple data
- Threads in step form a *group*: 8 here; on NVIDIA, 32, a *warp*

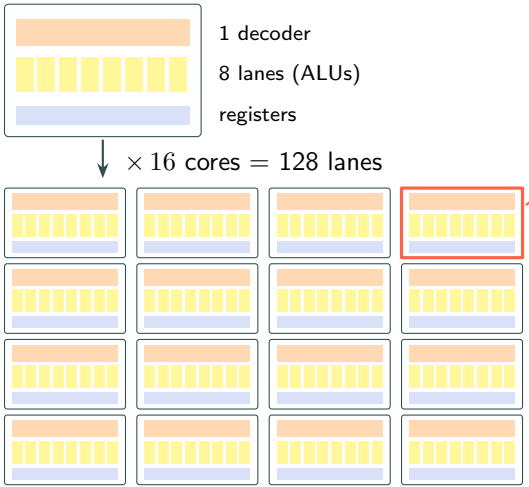
16 cores $\times$ 8 lanes: 128 pixels at once

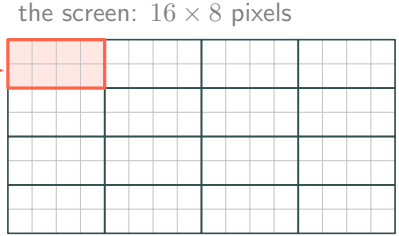
Ideas 1 and 2 together



16 cores $\times$ 8 lanes: 128 pixels at once

Ideas 1 and 2 together

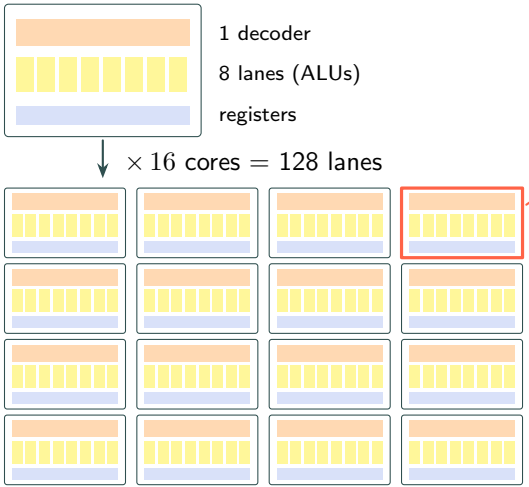




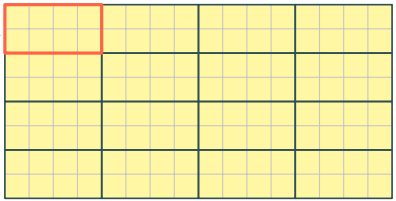
one core: one tile of 8 pixels

16 cores × 8 lanes: 128 pixels at once

Ideas 1 and 2 together



the screen: 16×8 pixels

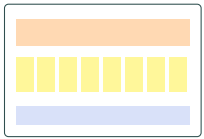


one core: one tile of 8 pixels

all 128 shaded in one tick

16 cores × 8 lanes: 128 pixels at once

Ideas 1 and 2 together



1 decoder: **step 2**

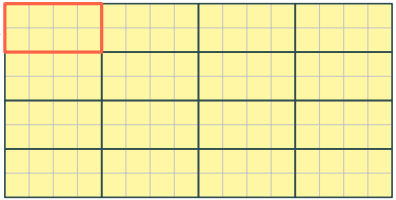
8 lanes (ALUs)

registers

$\times 16$ cores $= 128$ lanes



the screen: 16×8 pixels



one core: one tile of 8 pixels

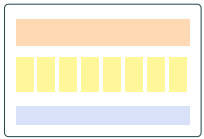
all 128 shaded in one tick

cores: each at its own step

lanes in one core: the same step

16 cores × 8 lanes: 128 pixels at once

Ideas 1 and 2 together



1 decoder: **step 2**

8 lanes (ALUs)

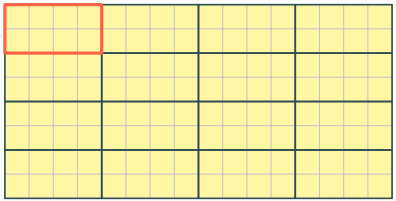
registers

$\times 16$ cores $= 128$ lanes



next: two problems, an `if` and memory

the screen: 16×8 pixels



one core: one tile of 8 pixels

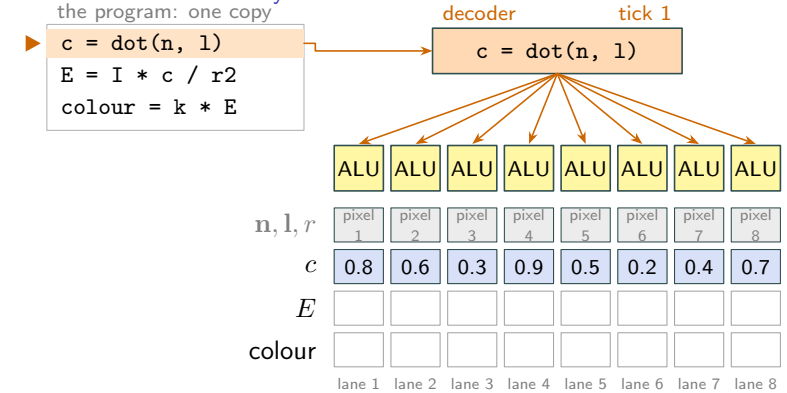
all 128 shaded in one tick

cores: each at its own step

lanes in one core: the same step

One line a tick, for all eight lanes

Before the `if`: what exactly is a lane?

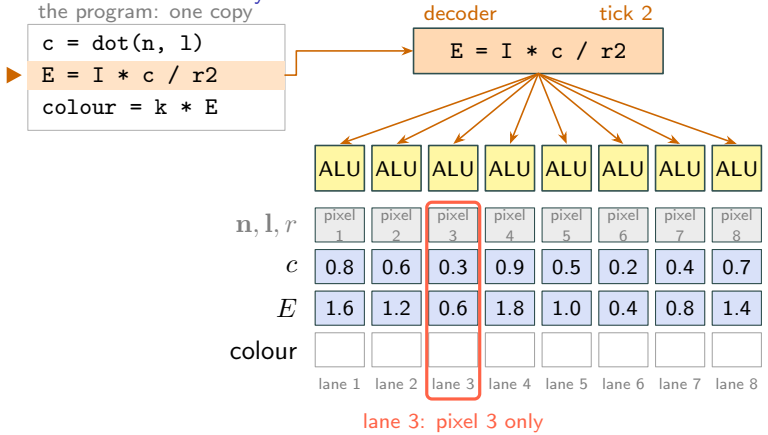


- Each tick the decoder reads one line
- All eight lanes run it at once

grey row: in each lane before tick 1

One line a tick, for all eight lanes

Before the `if`: what exactly is a lane?

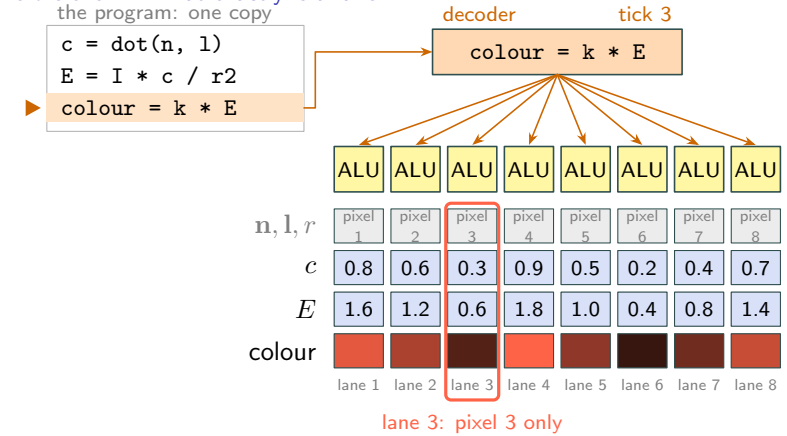


- Each tick the decoder reads one line
- All eight lanes run it at once
- A lane's column holds one pixel's numbers

grey row: in each lane before tick 1

One line a tick, for all eight lanes

Before the `if`: what exactly is a lane?

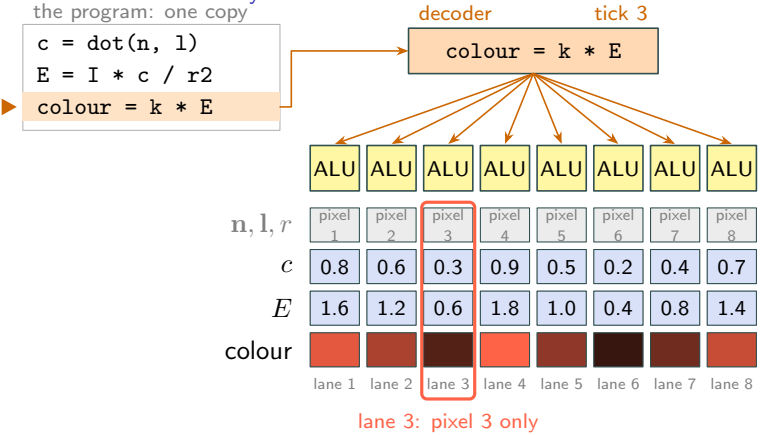


grey row: in each lane before tick 1

- Each tick the decoder reads one line
- All eight lanes run it at once
- A lane's column holds one pixel's numbers

One line a tick, for all eight lanes

Before the `if`: what exactly is a lane?



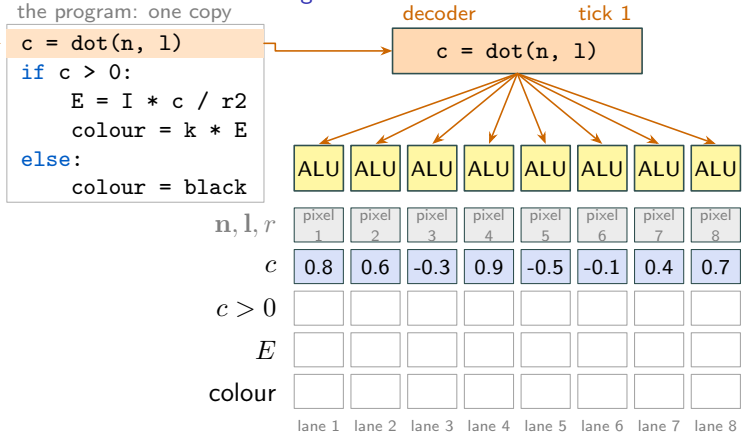
grey row: in each lane before tick 1

3 ticks, one row each; 8 numbers a row: $3 \times 8 = 24$

- Each tick the decoder reads one line
- All eight lanes run it at once
- A lane's column holds one pixel's numbers
- **Lanes carry data; instructions come only from the decoder**

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?

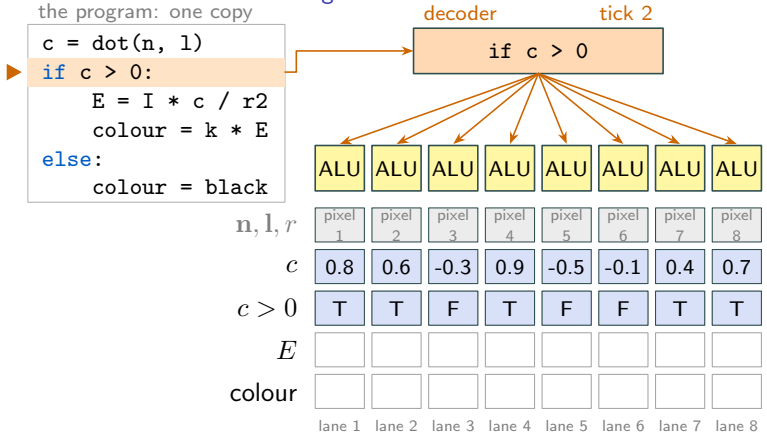


- All eight lanes share one instruction

yellow: working grey: masked, waiting

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?

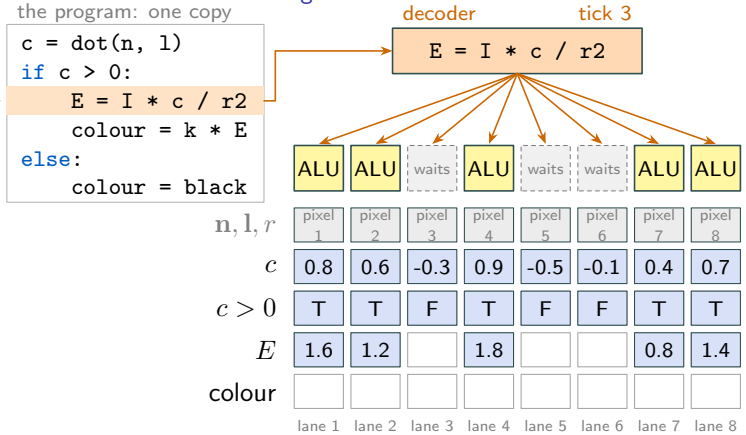


- All eight lanes share one instruction
- The test splits the lanes: 5 true, 3 false

yellow: working grey: masked, waiting

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?

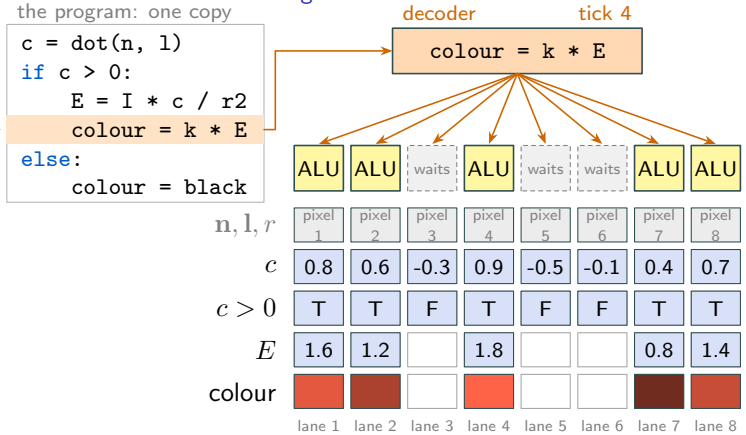


- All eight lanes share one instruction
- The test splits the lanes: 5 true, 3 false
- Each side runs while the other lanes wait

yellow: working grey: masked, waiting

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?



- All eight lanes share one instruction
- The test splits the lanes: 5 true, 3 false
- Each side runs while the other lanes wait

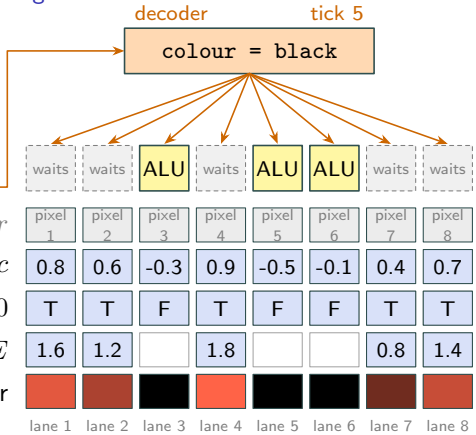
yellow: working grey: masked, waiting

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?

the program: one copy

```
c = dot(n, l)
if c > 0:
    E = I * c / r2
    colour = k * E
else:
    colour = black
```



- All eight lanes share one instruction
- The test splits the lanes: 5 true, 3 false
- Each side runs while the other lanes wait

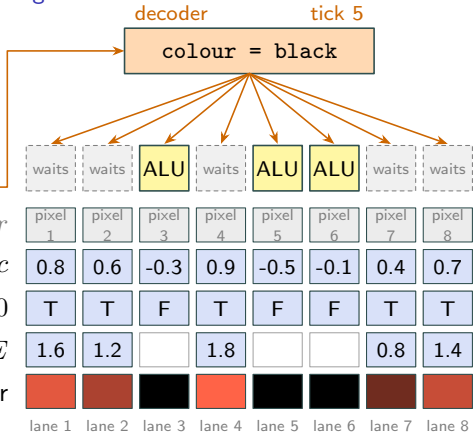
yellow: working grey: masked, waiting

A branch that only some lanes take: both sides run

Problem 1: what if the lanes disagree at an if?

the program: one copy

```
c = dot(n, l)
if c > 0:
    E = I * c / r2
    colour = k * E
else:
    colour = black
```



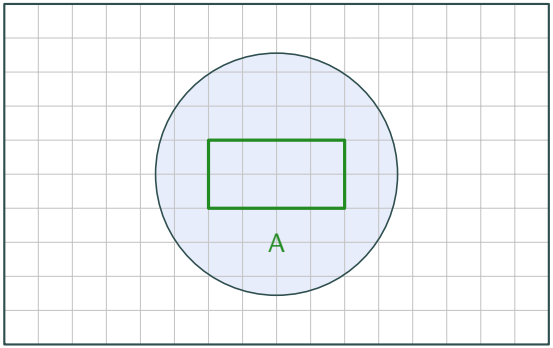
- All eight lanes share one instruction
- The test splits the lanes: 5 true, 3 false
- Each side runs while the other lanes wait
- Lanes splitting is called *divergence*: it costs the if-part plus the else-part

every lane spent 5 ticks; with no split: T lanes 4, F lanes 3

yellow: working grey: masked, waiting

Neighbours usually agree; edges split them

How often do neighbouring pixels disagree?

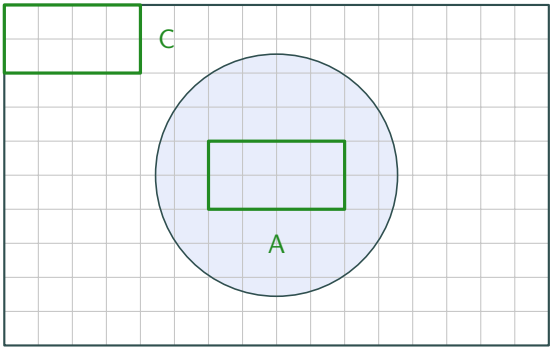


pixels; each outlined tile is one group of 8

- Tile A is all ball: one path

Neighbours usually agree; edges split them

How often do neighbouring pixels disagree?

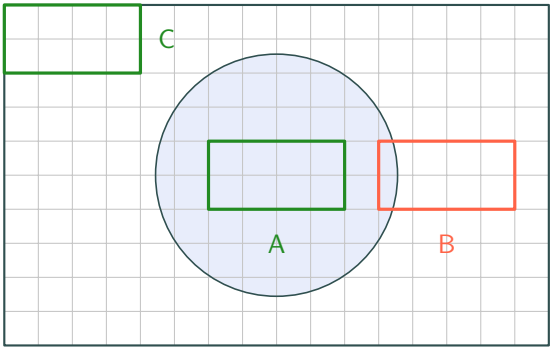


pixels; each outlined tile is one group of 8

- Tile A is all ball: one path
- Tile C is all sky: one path

Neighbours usually agree; edges split them

How often do neighbouring pixels disagree?

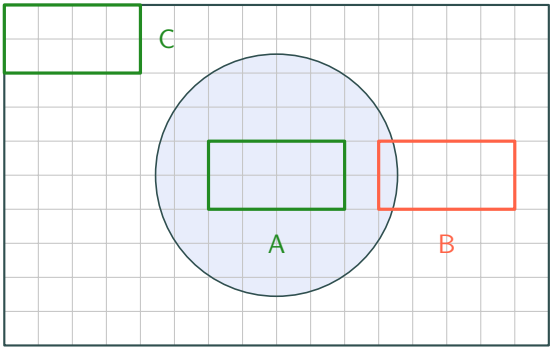


pixels; each outlined tile is one group of 8

- Tile A is all ball: one path
- Tile C is all sky: one path
- Tile B crosses the edge: it *diverges*

Neighbours usually agree; edges split them

How often do neighbouring pixels disagree?

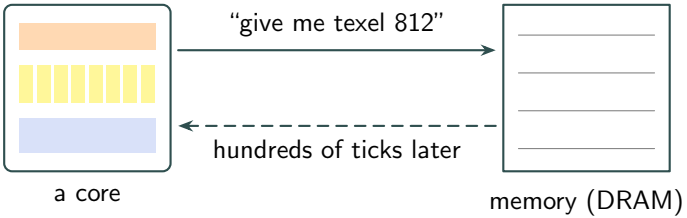


pixels; each outlined tile is one group of 8

- Tile A is all ball: one path
- Tile C is all sky: one path
- Tile B crosses the edge: it *diverges*
- Most tiles agree: the work is *coherent*

Memory is far: a read takes hundreds of ticks

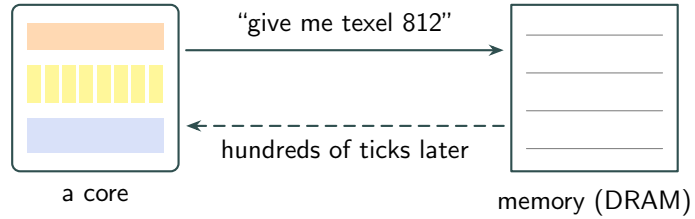
Problem 2: the lanes read memory, and memory is far



- A read takes hundreds of ticks

Memory is far: a read takes hundreds of ticks

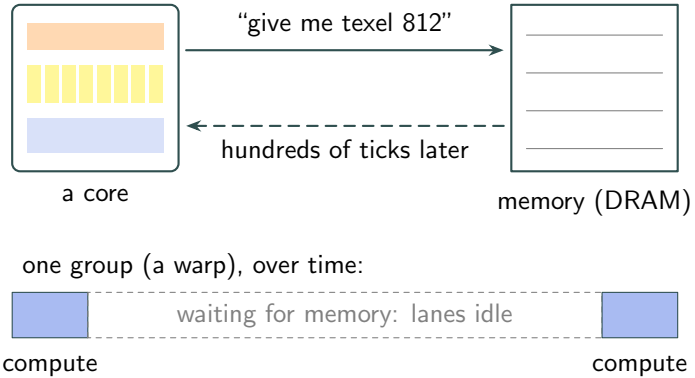
Problem 2: the lanes read memory, and memory is far



- A read takes hundreds of ticks
- Meanwhile the lanes sit idle

Memory is far: a read takes hundreds of ticks

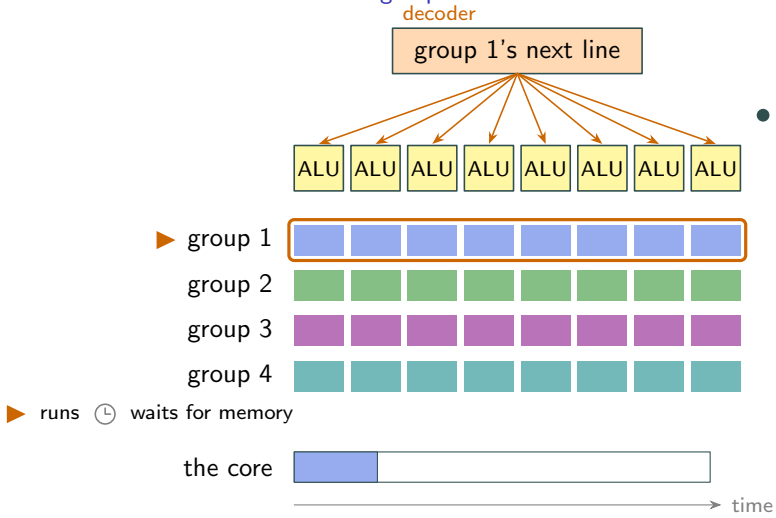
Problem 2: the lanes read memory, and memory is far



- A read takes hundreds of ticks
- Meanwhile the lanes sit idle
- Idea 1 threw out the big cache

Idea 3: while one group waits, run another

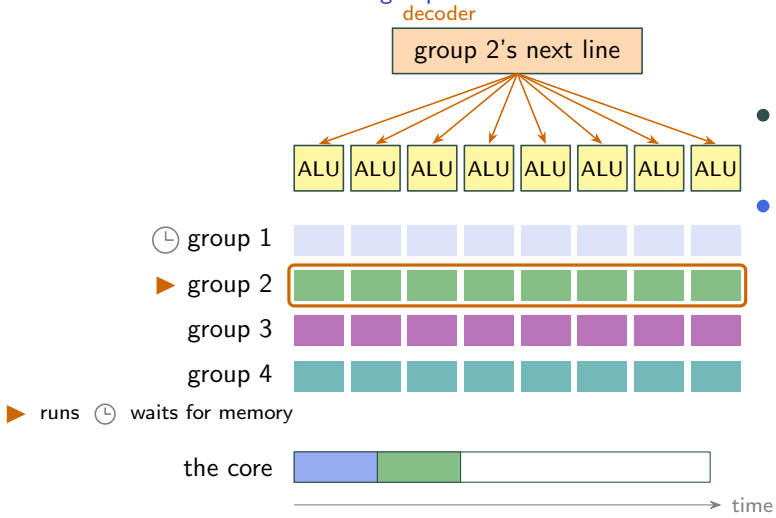
Answer: never let the core wait with the group



- One core holds four groups

Idea 3: while one group waits, run another

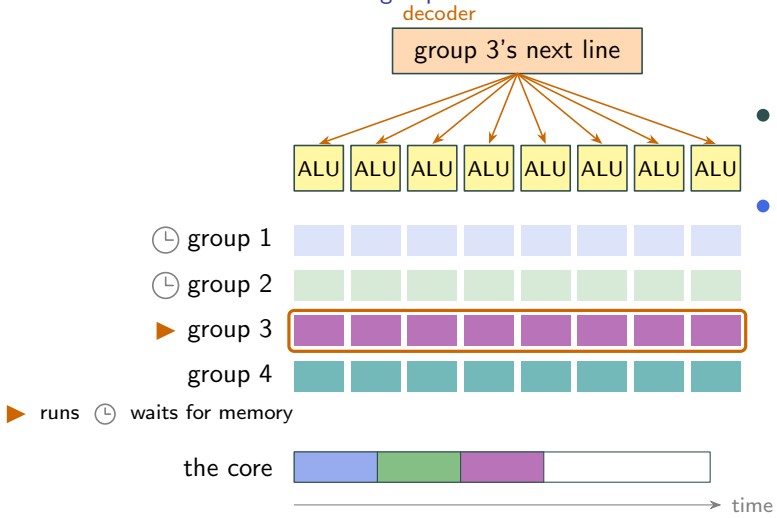
Answer: never let the core wait with the group



- One core holds four groups
- A waiting group steps aside

Idea 3: while one group waits, run another

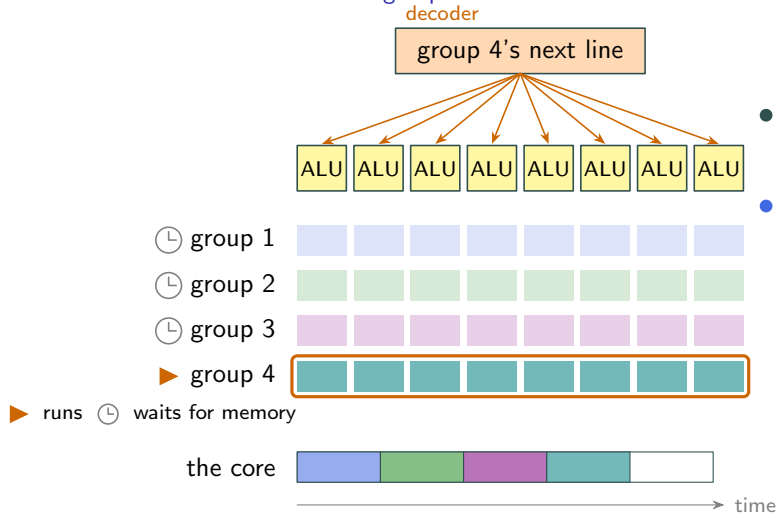
Answer: never let the core wait with the group



- One core holds four groups
- A waiting group steps aside

Idea 3: while one group waits, run another

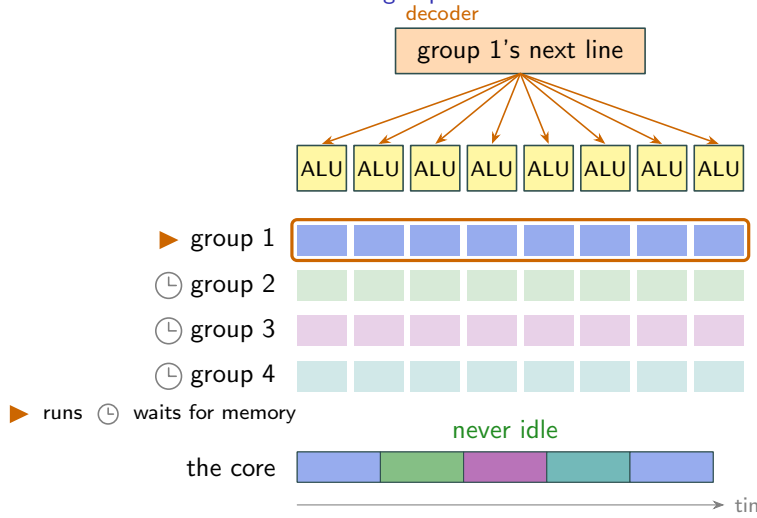
Answer: never let the core wait with the group



- One core holds four groups
- A waiting group steps aside

Idea 3: while one group waits, run another

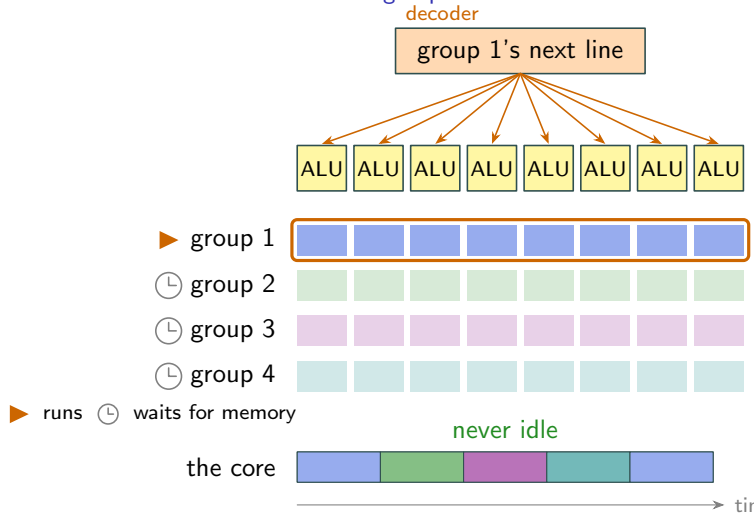
Answer: never let the core wait with the group



- One core holds four groups
- A waiting group steps aside
- The core is never idle

Idea 3: while one group waits, run another

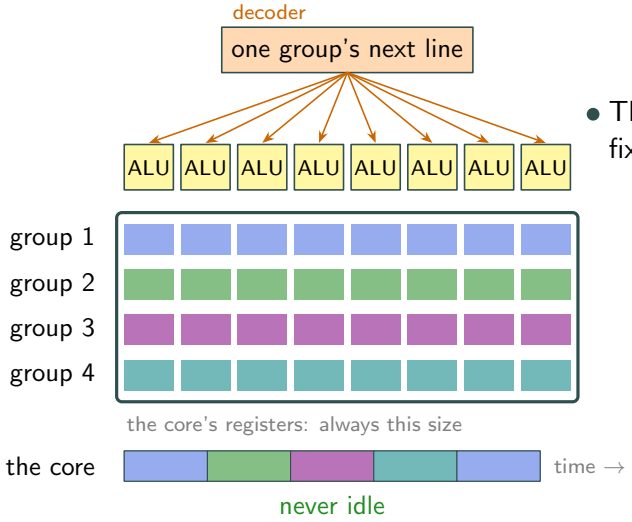
Answer: never let the core wait with the group



- One core holds four groups
- A waiting group steps aside
- The core is never idle
- The wait is hidden, not shortened

The price: each waiting group keeps its registers

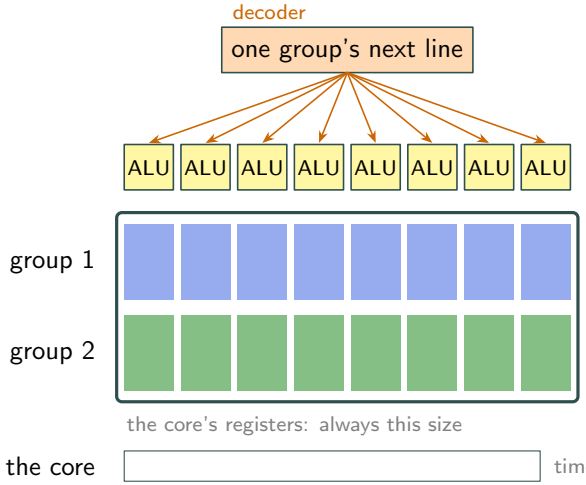
What does idea 3 cost?



- The registers are a box of fixed size

The price: each waiting group keeps its registers

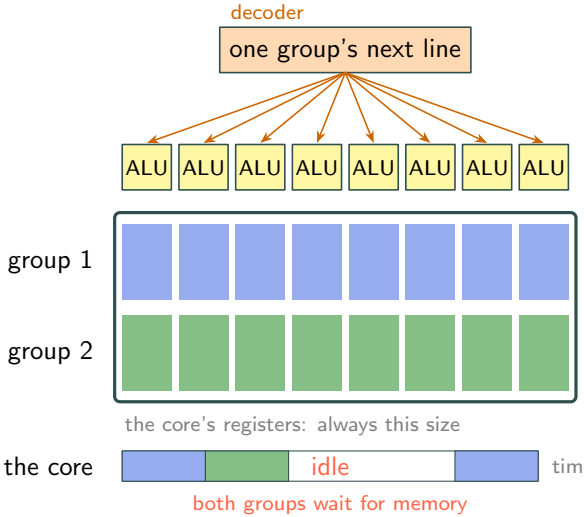
What does idea 3 cost?



- The registers are a box of fixed size
- More variables per pixel: fewer groups fit

The price: each waiting group keeps its registers

What does idea 3 cost?

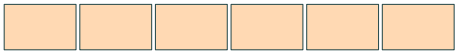


- The registers are a box of fixed size
- More variables per pixel: fewer groups fit
- Fewer groups: the waits show again

Built for throughput, not for latency

What have the three ideas built?

CPU core: one job at a time, each done soon



6 jobs done

each job: 1 unit of time

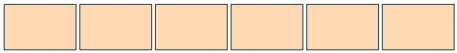


- Latency: how soon one job ends

Built for throughput, not for latency

What have the three ideas built?

CPU core: one job at a time, each done soon



6 jobs done

each job: 1 unit of time

GPU core: many jobs at once, each done late

24 jobs done

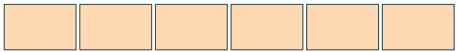
each job: 3 units of time

- Latency: how soon one job ends
- Throughput: how many jobs end

Built for throughput, not for latency

What have the three ideas built?

CPU core: one job at a time, each done soon



6 jobs done

each job: 1 unit of time

GPU core: many jobs at once, each done late

24 jobs done

each job: 3 units of time

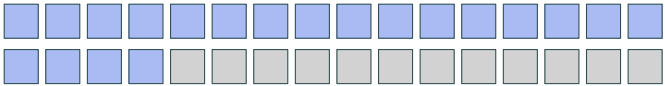


- Latency: how soon one job ends
- Throughput: how many jobs end
- A frame needs every pixel: throughput wins

Quick check 2: a warp that splits

Check: what a split costs one warp

one warp: 32 lanes, one instruction at a time



20 lanes: `if` side, 30 ticks

12 lanes: `else` side, 10 ticks

(a) How long does the warp take?

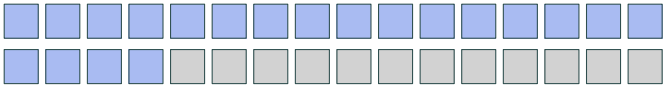
(b) And if all 32 took the `if` side?

(c) What share of the lanes' time was useful?

Quick check 2: a warp that splits

Check: what a split costs one warp

one warp: 32 lanes, one instruction at a time



20 lanes: `if` side, 30 ticks

12 lanes: `else` side, 10 ticks

(a) How long does the warp take?

$30 + 10 = 40$ ticks

(b) And if all 32 took the `if` side?

30 ticks

(c) What share of the lanes' time was useful?

$$\frac{20 \times 30 + 12 \times 10}{32 \times 40} = \frac{720}{1280} = 56\%$$

So far: a machine built for throughput

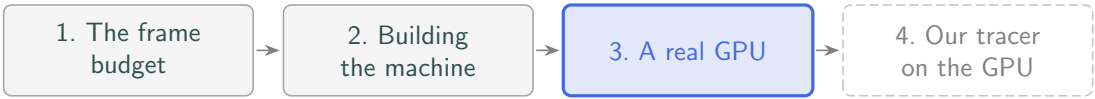
1. many slim cores
2. 32 lanes per instruction
3. many groups per core, to hide memory waits

- Fast when neighbouring lanes take the same path: *coherent* work
- Slow when lanes split at a branch, or too few groups hide the waits

Next question: does a real GPU look like this, and how fast is it?

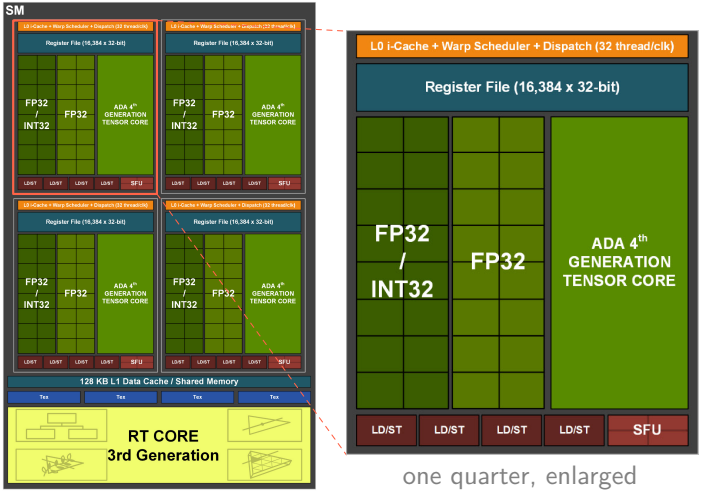
3. A real GPU

Does a real GPU look like our design?



One SM: four partitions of 32 lanes

NVIDIA's core, the SM (streaming multiprocessor): find ideas 2 and 3



one quarter, enlarged

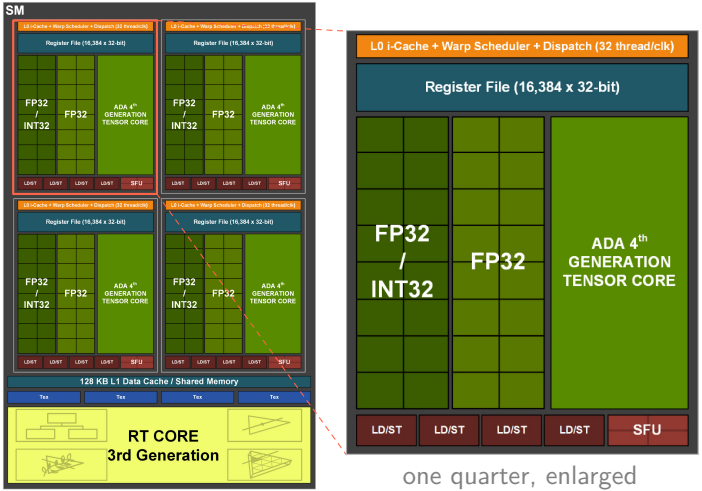
- 4 schedulers pick ready warps (idea 3)

SM = 4 quarters, each 32 lanes
quarter: holds 12 warps, runs 1 warp = 32 threads, one per lane
NVIDIA's "CUDA core" = 1 lane

NVIDIA Ada GPU Architecture whitepaper, Figure 5; see images/CREDITS.txt

One SM: four partitions of 32 lanes

NVIDIA's core, the SM (streaming multiprocessor): find ideas 2 and 3



one quarter, enlarged

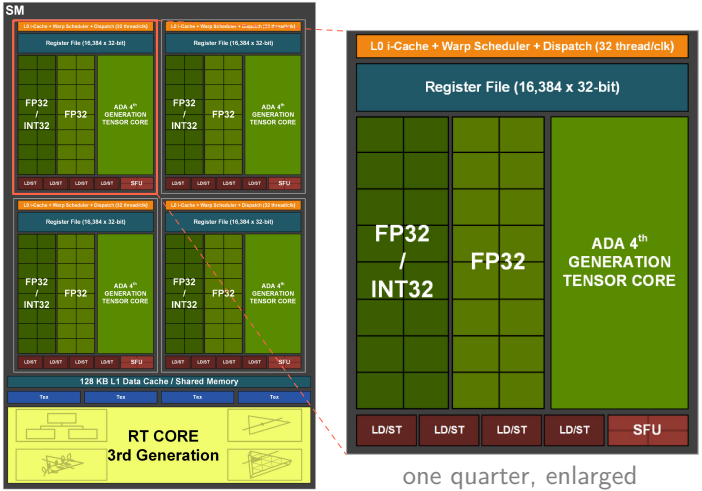
- 4 schedulers pick ready warps (idea 3)
- $4 \times 32 = 128$ lanes (idea 2)

SM = 4 quarters, each 32 lanes
quarter: holds 12 warps, runs 1 warp = 32 threads, one per lane
NVIDIA's "CUDA core" = 1 lane

NVIDIA Ada GPU Architecture whitepaper, Figure 5; see images/CREDITS.txt

One SM: four partitions of 32 lanes

NVIDIA's core, the SM (streaming multiprocessor): find ideas 2 and 3



one quarter, enlarged

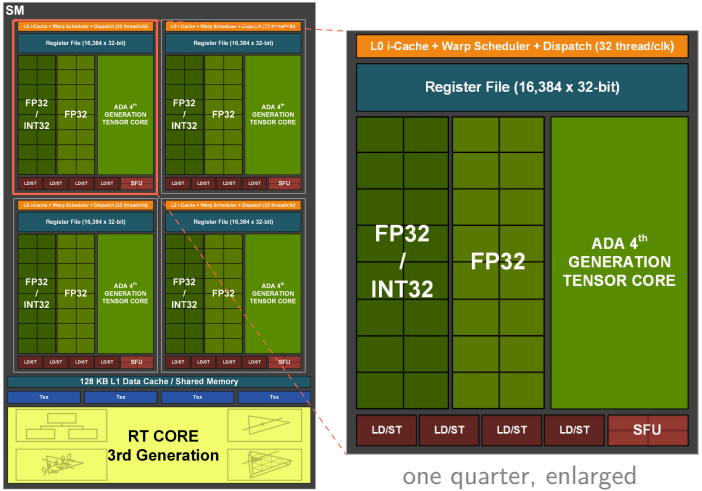
- 4 schedulers pick ready warps (idea 3)
- $4 \times 32 = 128$ lanes (idea 2)
- Holds 48 warps, runs 4 at a time (idea 3)

SM = 4 quarters, each 32 lanes
quarter: holds 12 warps, runs 1 warp = 32 threads, one per lane
NVIDIA's "CUDA core" = 1 lane

NVIDIA Ada GPU Architecture whitepaper, Figure 5; see images/CREDITS.txt

One SM: four partitions of 32 lanes

NVIDIA's core, the SM (streaming multiprocessor): find ideas 2 and 3



one quarter, enlarged

- 4 schedulers pick ready warps (idea 3)
- $4 \times 32 = 128$ lanes (idea 2)
- Holds 48 warps, runs 4 at a time (idea 3)
- Texture units and the RT core are circuits

SM = 4 quarters, each 32 lanes
quarter: holds 12 warps, runs 1 warp = 32 threads, one per lane
NVIDIA's "CUDA core" = 1 lane

NVIDIA Ada GPU Architecture whitepaper, Figure 5; see images/CREDITS.txt

The chip: 144 SMs around one shared cache

Zoom out to the whole chip: the SM, copied (idea 1)



- 12 clusters $\times$ 12 SMs

NVIDIA Ada GPU Architecture whitepaper, Figure 1; see images/CREDITS.txt

The chip: 144 SMs around one shared cache

Zoom out to the whole chip: the SM, copied (idea 1)



- 12 clusters $\times$ 12 SMs
- L2 cache in the middle

NVIDIA Ada GPU Architecture whitepaper, Figure 1; see images/CREDITS.txt

The chip: 144 SMs around one shared cache

Zoom out to the whole chip: the SM, copied (idea 1)

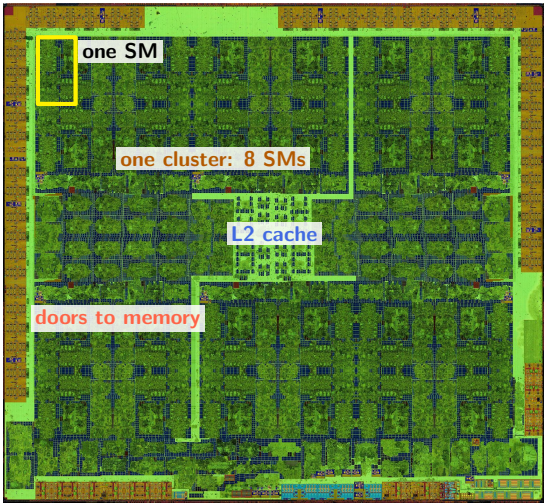


- 12 clusters $\times$ 12 SMs
- L2 cache in the middle
- Memory controllers: the sides

NVIDIA Ada GPU Architecture whitepaper, Figure 1; see images/CREDITS.txt

On the die: one block, copied many times

The same plan in a photograph; compare the CPU die of slide 16

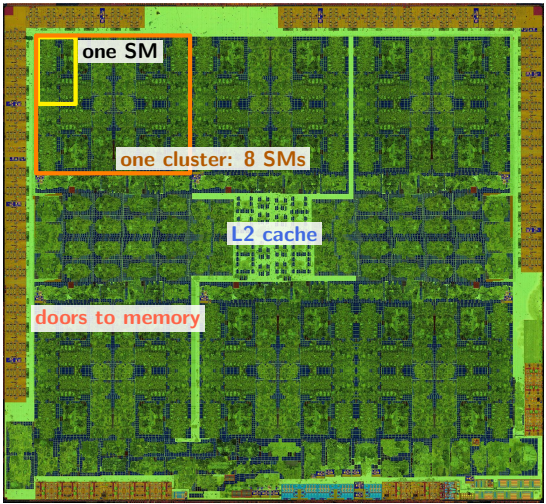


NVIDIA TU104 (GeForce RTX 2080): Fritzchens Fritz, CC0; see images/CREDITS.txt

- One tile, the SM, repeats in rows

On the die: one block, copied many times

The same plan in a photograph; compare the CPU die of slide 16

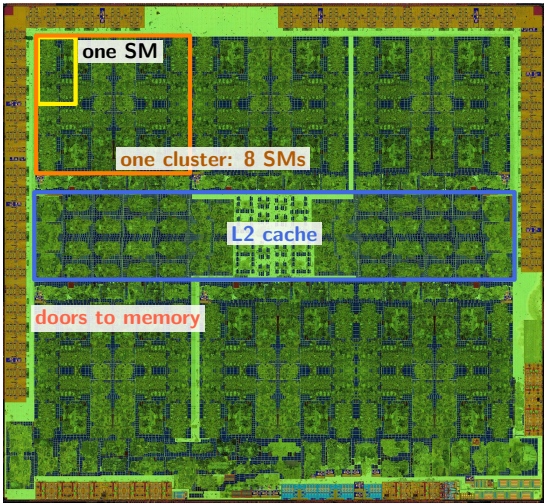


NVIDIA TU104 (GeForce RTX 2080): Fritzchens Fritz, CC0; see images/CREDITS.txt

- One tile, the SM, repeats in rows
- 8 SMs make a cluster; this chip has 6

On the die: one block, copied many times

The same plan in a photograph; compare the CPU die of slide 16

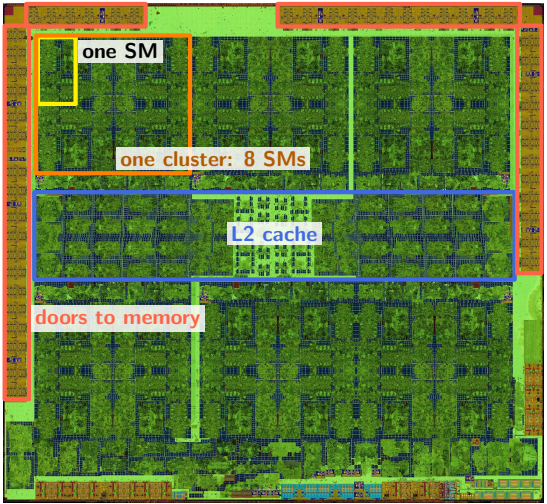


NVIDIA TU104 (GeForce RTX 2080): Fritzchens Fritz, CC0; see images/CREDITS.txt

- One tile, the SM, repeats in rows
- 8 SMs make a cluster; this chip has 6
- One cache in the middle, shared by all SMs

On the die: one block, copied many times

The same plan in a photograph; compare the CPU die of slide 16

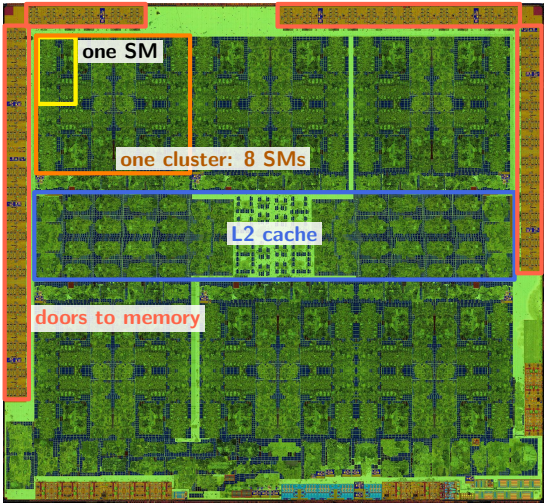


NVIDIA TU104 (GeForce RTX 2080): Fritzchens Fritz, CC0; see images/CREDITS.txt

- One tile, the SM, repeats in rows
- 8 SMs make a cluster; this chip has 6
- One cache in the middle, shared by all SMs
- Doors to memory round the edge

On the die: one block, copied many times

The same plan in a photograph; compare the CPU die of slide 16



NVIDIA TU104 (GeForce RTX 2080): Fritzchens Fritz, CC0; see images/CREDITS.txt

- One tile, the SM, repeats in rows
- 8 SMs make a cluster; this chip has 6
- One cache in the middle, shared by all SMs
- Doors to memory round the edge
- **Most of the area computes**

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

$$128 \text{ SMs} \times 128 \text{ lanes} = 16{,}384 \text{ lanes}$$

RTX 4090

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

128 SMs $\times$ 128 lanes $= 16{,}384$ lanes RTX 4090

$\times$ 2.52 billion ticks a second

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

128 SMs $\times$ 128 lanes $= 16{,}384$ lanes RTX 4090

$\times$ 2.52 billion ticks a second

$\times$ 2: one multiply-add is 2 operations

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

128 SMs $\times$ 128 lanes $= 16{,}384$ lanes RTX 4090

$\times$ 2.52 billion ticks a second

$\times$ 2: one multiply-add is 2 operations

$=$ 82.6 trillion operations a second

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

128 SMs $\times$ 128 lanes $= 16{,}384$ lanes RTX 4090

$\times$ 2.52 billion ticks a second

$\times$ 2: one multiply-add is 2 operations

$=$ 82.6 trillion operations a second

$$\frac{82.6 \times 10^{12}}{60 \times 2{,}073{,}600} \approx 660{,}000 \text{ per pixel per frame}$$

Count it: 82.6 trillion operations a second

How much arithmetic does the whole chip give each pixel?

128 SMs $\times$ 128 lanes $= 16{,}384$ lanes RTX 4090

$\times$ 2.52 billion ticks a second

$\times$ 2: one multiply-add is 2 operations

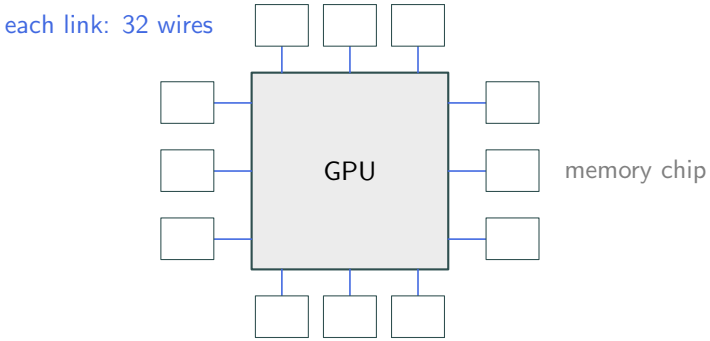
$=$ 82.6 trillion operations a second

$$\frac{82.6 \times 10^{12}}{60 \times 2{,}073{,}600} \approx 660{,}000 \text{ per pixel per frame}$$

- one CPU core (slide 9): 40 ticks per pixel

Memory: a road 384 wires wide

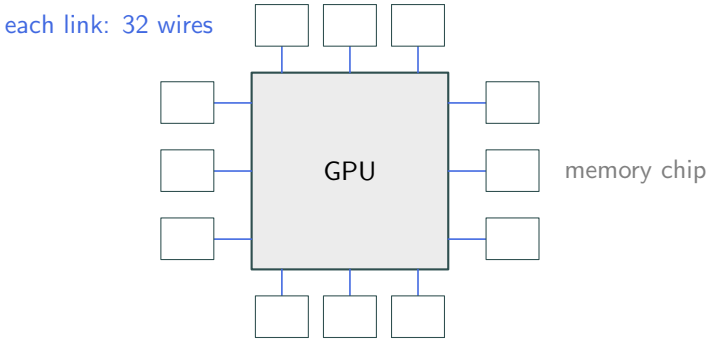
Can memory keep 16,384 lanes fed?



- $12 \times 32 = 384$ wires at once

Memory: a road 384 wires wide

Can memory keep 16,384 lanes fed?

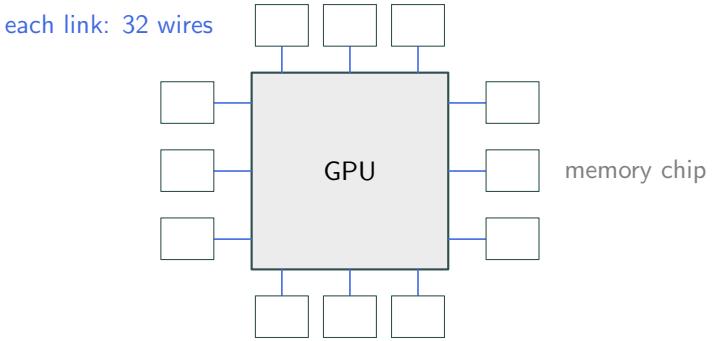


- $12 \times 32 = 384$ wires at once
- About 1 TB every second

GPU: 384 bits $\times$ 21 Gbit/s $/ 8 = 1{,}008$ GB/s

Memory: a road 384 wires wide

Can memory keep 16,384 lanes fed?



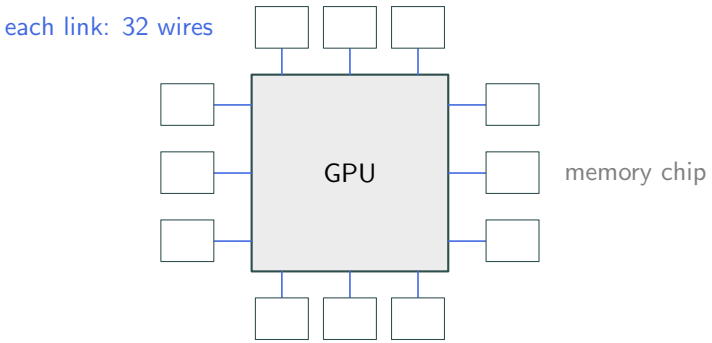
- $12 \times 32 = 384$ wires at once
- About 1 TB every second

GPU: 384 bits $\times$ 21 Gbit/s $/ 8 = 1{,}008$ GB/s

CPU: 128 bits $\times$ 5.6 Gbit/s $/ 8 = 90$ GB/s

Memory: a road 384 wires wide

Can memory keep 16,384 lanes fed?



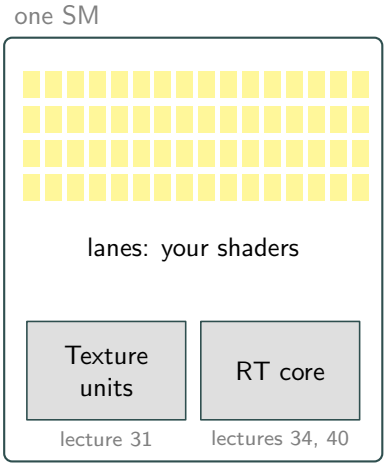
- $12 \times 32 = 384$ wires at once
- About 1 TB every second
- Often memory, not arithmetic, is the limit

GPU: 384 bits $\times$ 21 Gbit/s $/ 8 = 1{,}008$ GB/s

CPU: 128 bits $\times$ 5.6 Gbit/s $/ 8 = 90$ GB/s

Circuits beside the programmable lanes

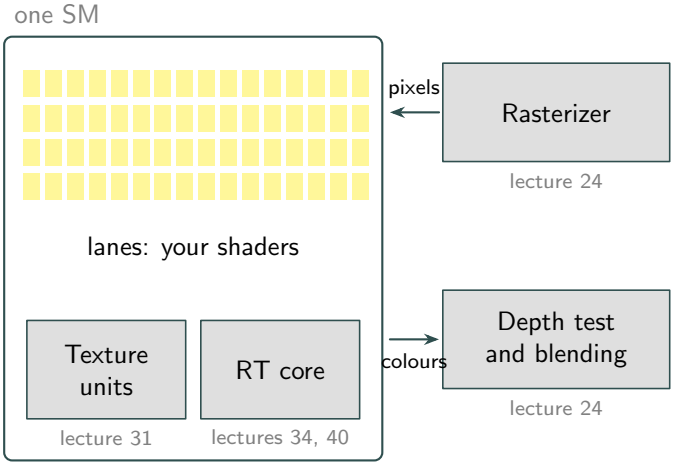
Not all of the chip is lanes



- Jobs done the same way every frame get circuits

Circuits beside the programmable lanes

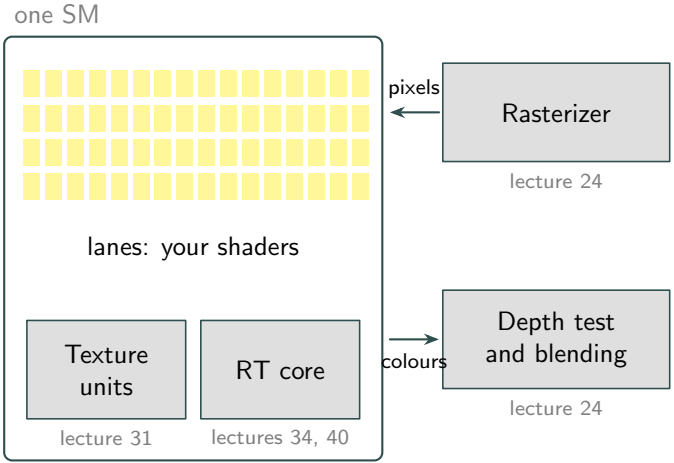
Not all of the chip is lanes



- Jobs done the same way every frame get circuits

Circuits beside the programmable lanes

Not all of the chip is lanes



- Jobs done the same way every frame get circuits
- A circuit is faster and smaller than a program
- Next: how the chip came to be this mix

First: a fixed circuit for each step

How it began: graphics chips were nothing but circuits



1993 SGI Onyx

a cabinet of boards, for studios

photographs: see images/CREDITS.txt

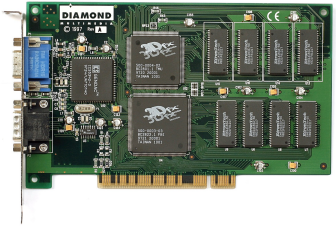
First: a fixed circuit for each step

How it began: graphics chips were nothing but circuits



1993 SGI Onyx

a cabinet of boards, for studios



1996 3dfx Voodoo

triangles and textures, in a PC

photographs: see images/CREDITS.txt

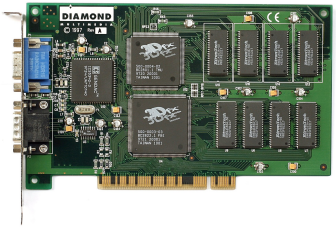
First: a fixed circuit for each step

How it began: graphics chips were nothing but circuits



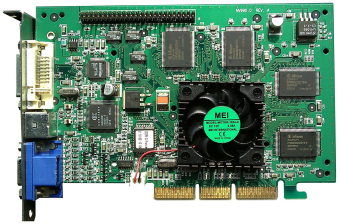
1993 SGI Onyx

a cabinet of boards, for studios



1996 3dfx Voodoo

triangles and textures, in a PC



1999 NVIDIA GeForce 256

sold as "the first GPU"

photographs: see images/CREDITS.txt

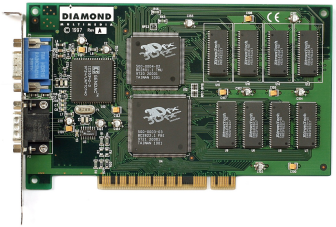
First: a fixed circuit for each step

How it began: graphics chips were nothing but circuits



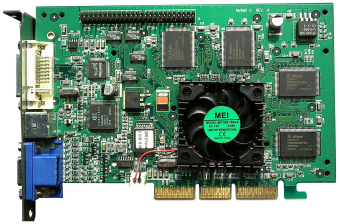
1993 SGI Onyx

a cabinet of boards, for studios



1996 3dfx Voodoo

triangles and textures, in a PC



1999 NVIDIA GeForce 256

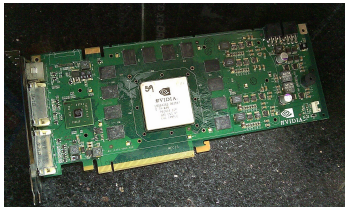
sold as "the first GPU"

- Each step was a circuit: fast, but not flexible

photographs: see images/CREDITS.txt

Then: the circuits became programmable

Games wanted lighting that no circuit had built in



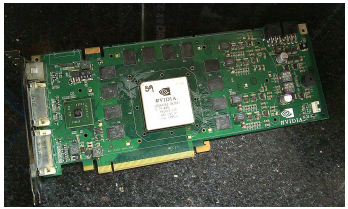
2006 GeForce 8800
one kind of core for every stage

- **2001** GeForce 3: the first small programs, *shaders*

photographs: see images/CREDITS.txt

Then: the circuits became programmable

Games wanted lighting that no circuit had built in



2006 GeForce 8800
one kind of core for every stage



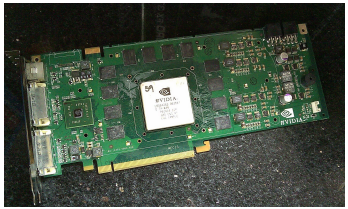
2018 GeForce RTX 2080
RT cores: ray tests in circuits

- **2001** GeForce 3: the first small programs, *shaders*

photographs: see images/CREDITS.txt

Then: the circuits became programmable

Games wanted lighting that no circuit had built in



2006 GeForce 8800
one kind of core for every stage



2018 GeForce RTX 2080
RT cores: ray tests in circuits



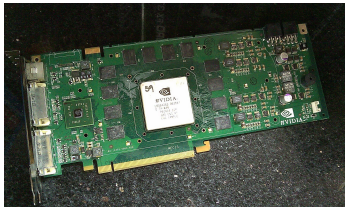
2022 GeForce RTX 4090
16,384 lanes (slide 35)

- **2001** GeForce 3: the first small programs, *shaders*

photographs: see images/CREDITS.txt

Then: the circuits became programmable

Games wanted lighting that no circuit had built in



2006 GeForce 8800
one kind of core for every stage



2018 GeForce RTX 2080
RT cores: ray tests in circuits



2022 GeForce RTX 4090
16,384 lanes (slide 35)

- **2001** GeForce 3: the first small programs, *shaders*
- Programs where work varies

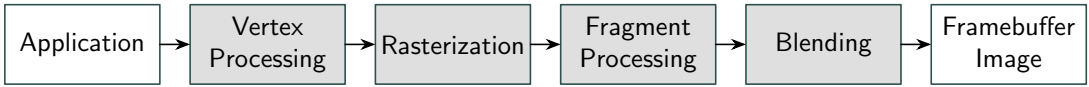
photographs: see images/CREDITS.txt

Some stages run your program, some are circuits

The result today: the textbook's pipeline, box by box

the textbook's Figure 17.1, laid on its side

1999: fixed function



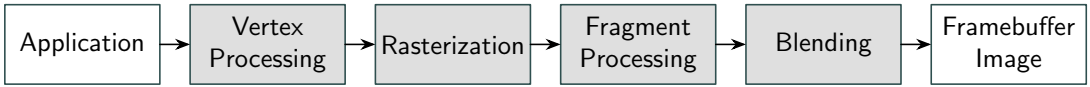
- Grey: a circuit with settings

Some stages run your program, some are circuits

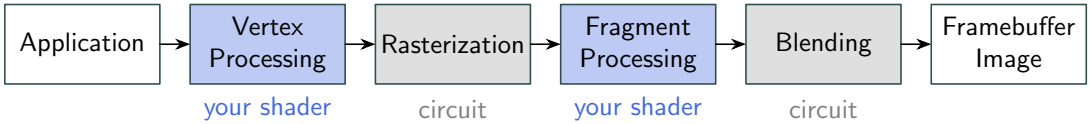
The result today: the textbook's pipeline, box by box

the textbook's Figure 17.1, laid on its side

1999: fixed function



today: programmable



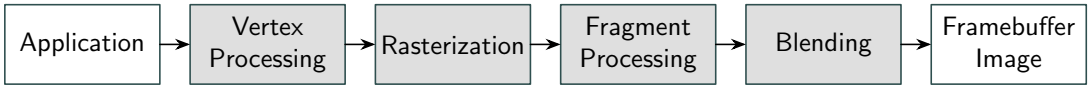
- Grey: a circuit with settings
- Blue: your own program

Some stages run your program, some are circuits

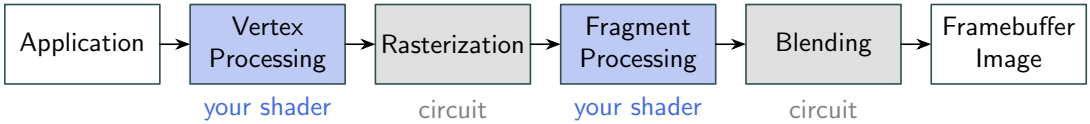
The result today: the textbook's pipeline, box by box

the textbook's Figure 17.1, laid on its side

1999: fixed function



today: programmable

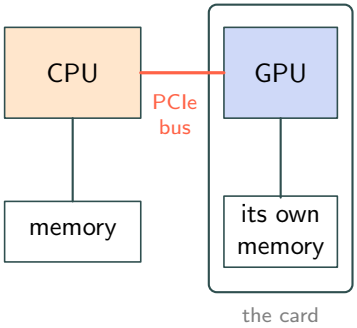


- Grey: a circuit with settings
- Blue: your own program
- Lecture 24: the trip

Your laptop: one chip, shared memory

And the GPU your setup check found?

discrete: a desktop with a card

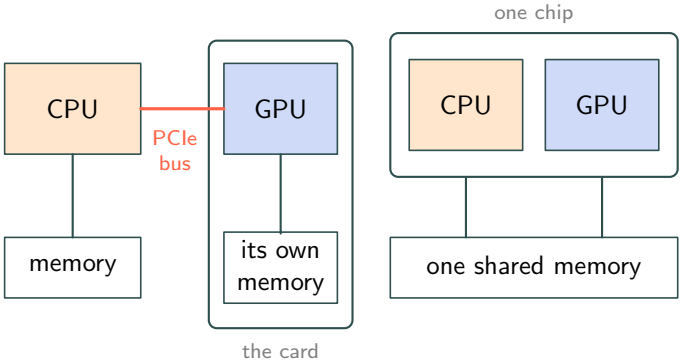


- Discrete: own memory, across a bus

Your laptop: one chip, shared memory

And the GPU your setup check found?

discrete: a desktop with a card **integrated:** most laptops, phones

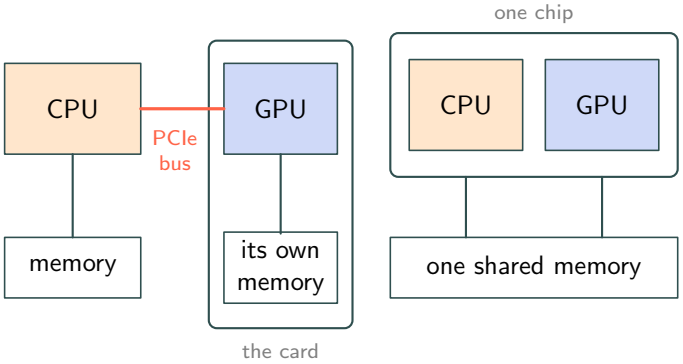


- Discrete: own memory, across a bus
- Integrated: one chip, one memory

Your laptop: one chip, shared memory

And the GPU your setup check found?

discrete: a desktop with a card **integrated:** most laptops, phones



- Discrete: own memory, across a bus
- Integrated: one chip, one memory
- Same three ideas, fewer cores

So far: the three ideas, in silicon

$$128 \text{ SMs} \times 128 \text{ lanes} = 16{,}384 \text{ lanes}$$

$$82.6 \text{ trillion operations a second} \qquad 1 \text{ TB/s}$$

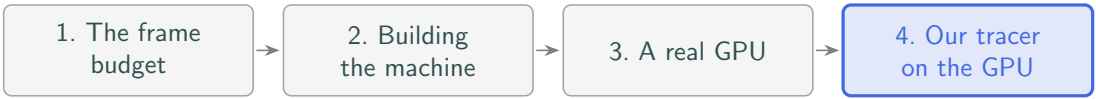
plus circuits for the jobs every frame does the same way:

rasterizer, texture units, depth and blend, RT cores

Next question: can our Part A ray tracer simply move onto it?

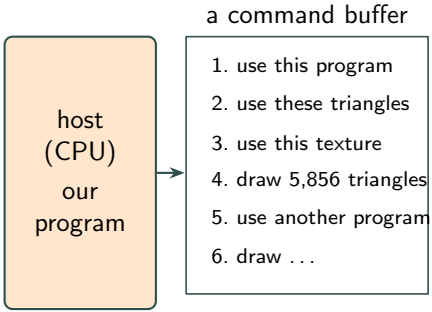
4. Our tracer on the GPU

Can our ray tracer move onto it, and what must Part B fake?



The host records commands; the device runs them

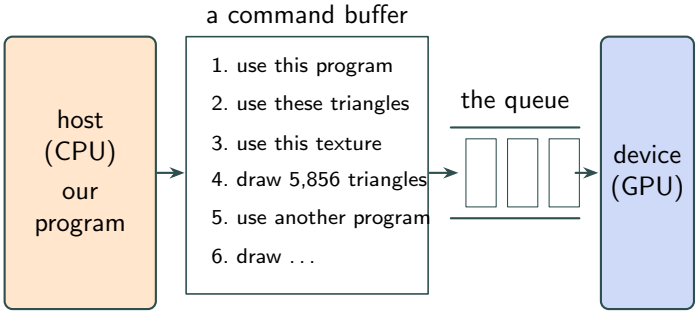
Before moving anything: how does our program reach the GPU?



- The host (CPU) writes a list of commands

The host records commands; the device runs them

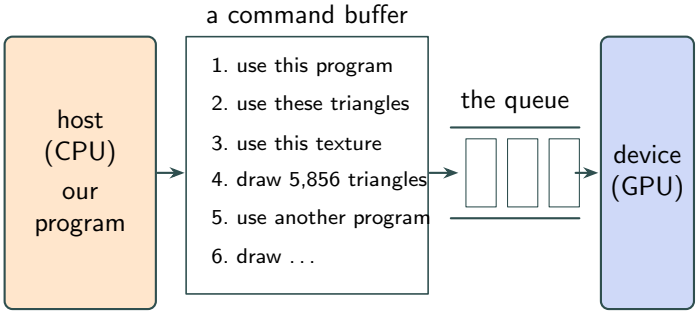
Before moving anything: how does our program reach the GPU?



- The host (CPU) writes a list of commands
- The device (GPU) works through the queue

The host records commands; the device runs them

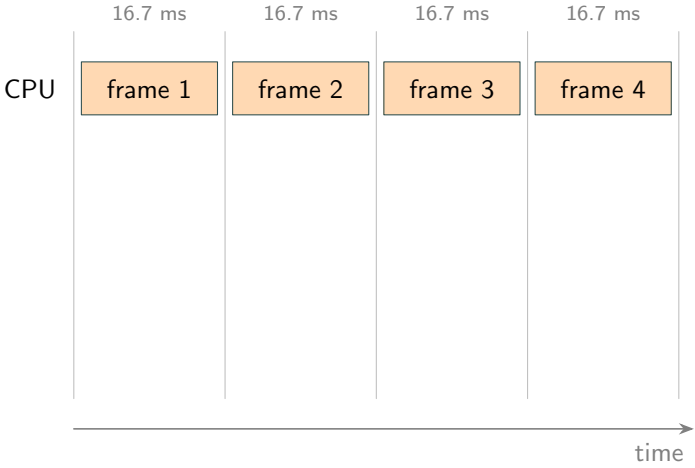
Before moving anything: how does our program reach the GPU?



- The host (CPU) writes a list of commands
- The device (GPU) works through the queue
- Neither waits for the other

Two machines, working one frame apart

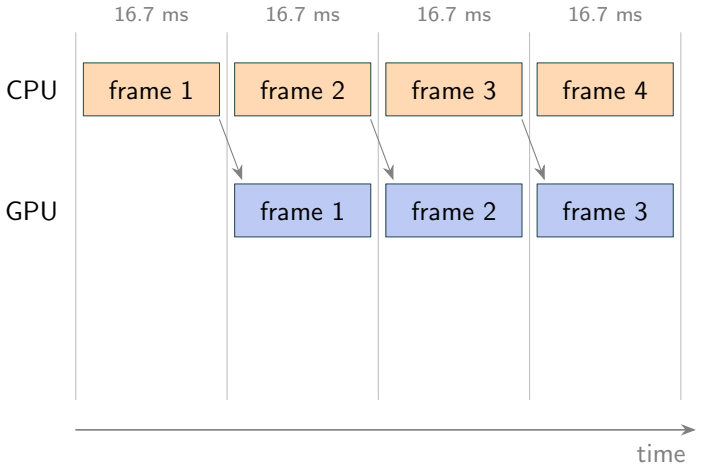
The same queue over time



- CPU prepares frame $N + 1$

Two machines, working one frame apart

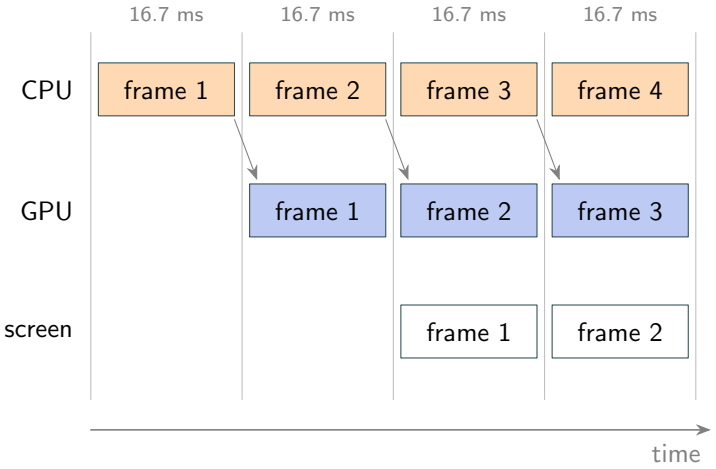
The same queue over time



- CPU prepares frame $N + 1$
- GPU draws frame N

Two machines, working one frame apart

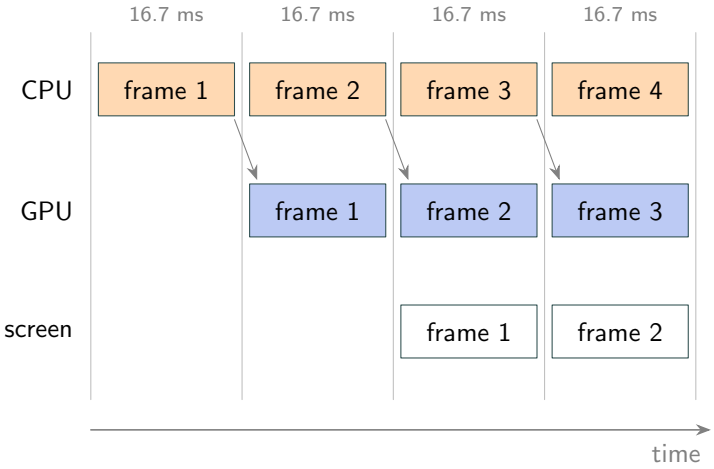
The same queue over time



- CPU prepares frame $N + 1$
- GPU draws frame N

Two machines, working one frame apart

The same queue over time



- CPU prepares frame $N + 1$
- GPU draws frame N
- Each machine must fit in 16.7 ms

No recursion: `trace` must become a loop

Moving `trace` across, obstacle 1: it calls itself

lecture 20, slide 33: `trace` calls itself

```
d = rec.normal + random_unit_vector()
return material.albedo * trace(Ray(x, d), depth - 1)
```

- A shader may not call itself

No recursion: `trace` must become a loop

Moving `trace` across, obstacle 1: it calls itself

lecture 20, slide 33: `trace` calls itself

```
d = rec.normal + random_unit_vector()
return material.albedo * trace(Ray(x, d), depth - 1)
```

the same path as a loop (chalk and glowing surfaces only):

```
weight = Vec3(1, 1, 1)
for k in range(depth):
    rec = scene.hit(ray)
    if rec.t == math.inf:
        return weight * sky
    material = rec.surface.material
    if isinstance(material, DiffuseLight):
        return weight * material.emission
    x = ray.evaluate(rec.t)
    d = rec.normal + random_unit_vector()
    weight = weight * material.albedo
    ray = Ray(x, d)
return Vec3(0, 0, 0)
```

- A shader may not call itself
- The cure: carry a weight, replace the ray

No recursion: `trace` must become a loop

Moving `trace` across, obstacle 1: it calls itself

lecture 20, slide 33: `trace` calls itself

```
d = rec.normal + random_unit_vector()
return material.albedo * trace(Ray(x, d), depth - 1)
```

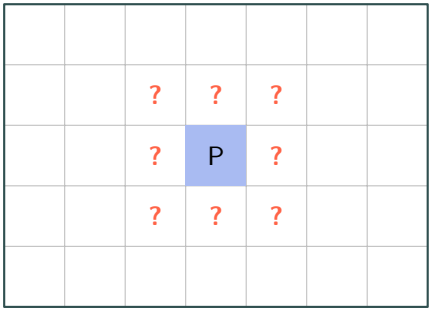
the same path as a loop (chalk and glowing surfaces only):

```
weight = Vec3(1, 1, 1)
for k in range(depth):
    rec = scene.hit(ray)
    if rec.t == math.inf:
        return weight * sky
    material = rec.surface.material
    if isinstance(material, DiffuseLight):
        return weight * material.emission
    x = ray.evaluate(rec.t)
    d = rec.normal + random_unit_vector()
    weight = weight * material.albedo
    ray = Ray(x, d)
return Vec3(0, 0, 0)
```

- A shader may not call itself
- The cure: carry a weight, replace the ray
- Lectures 27 and 40 use this loop

No reading a neighbour's answer in the same pass

Obstacle 2: real time borrows from neighbours (slide 3), and that needs care

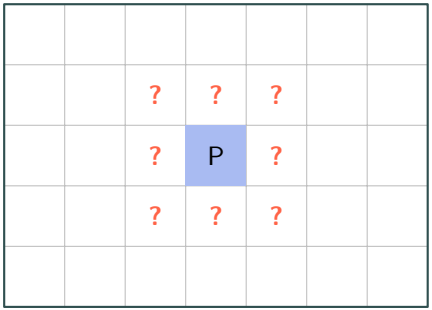


P wants a blur of its neighbours

- Pixels finish in no fixed order

No reading a neighbour's answer in the same pass

Obstacle 2: real time borrows from neighbours (slide 3), and that needs care



P wants a blur of its neighbours

pass 1: draw the picture, keep it

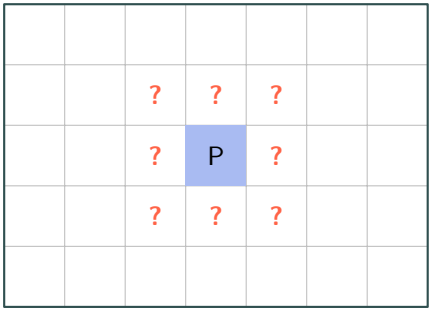
↓

pass 2: each pixel reads its neighbours

- Pixels finish in no fixed order
- Need a neighbour? Draw twice

No reading a neighbour's answer in the same pass

Obstacle 2: real time borrows from neighbours (slide 3), and that needs care



P wants a blur of its neighbours

pass 1: draw the picture, keep it

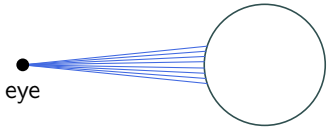
pass 2: each pixel reads its neighbours

- Pixels finish in no fixed order
- Need a neighbour? Draw twice
- Lecture 35: blur and bloom work this way

Why the secondary ray is so dear

Obstacle 3, the big one: what bounce rays do to the lanes

first rays

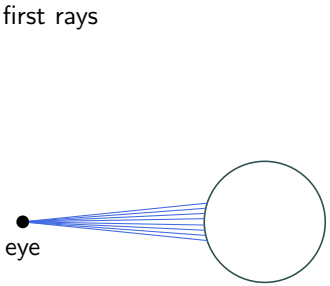


8 lanes: one ball, one path

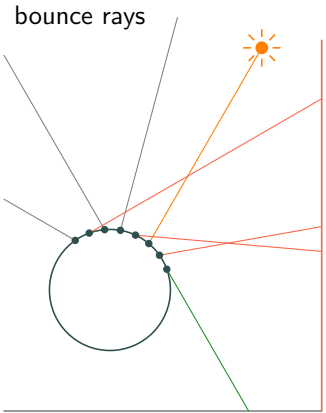
- First rays: neighbouring lanes agree

Why the secondary ray is so dear

Obstacle 3, the big one: what bounce rays do to the lanes



8 lanes: one ball, one path

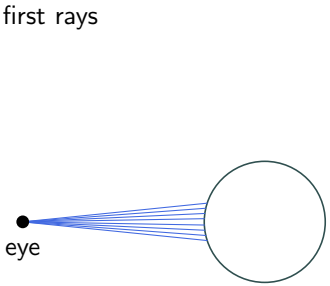


8 lanes: 4 places, 4 paths, scattered reads

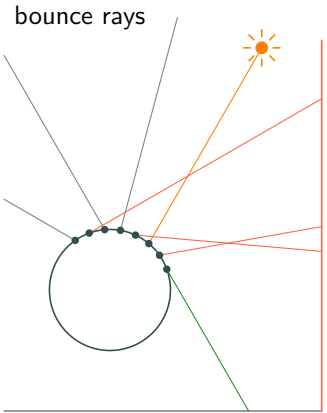
- First rays: neighbouring lanes agree
- Bounces: every lane differs

Why the secondary ray is so dear

Obstacle 3, the big one: what bounce rays do to the lanes



8 lanes: one ball, one path

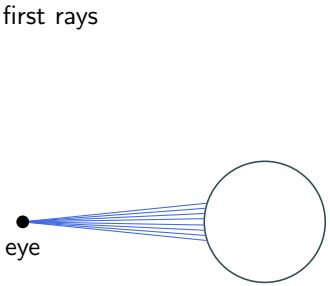


8 lanes: 4 places, 4 paths, scattered reads

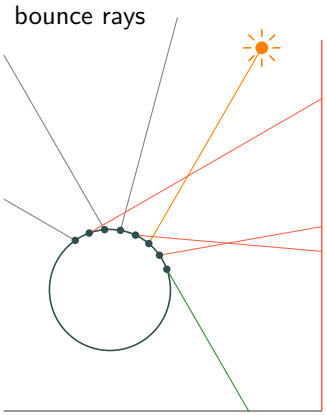
- First rays: neighbouring lanes agree
- Bounces: every lane differs
- Both weak spots of part two at once

Why the secondary ray is so dear

Obstacle 3, the big one: what bounce rays do to the lanes



8 lanes: one ball, one path

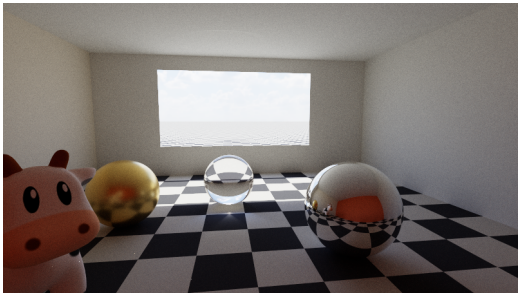


8 lanes: 4 places, 4 paths, scattered reads

- First rays: neighbouring lanes agree
- Bounces: every lane differs
- Both weak spots of part two at once
- Part B must answer without these rays

Where 16 ms breaks: light that bounces

What bounced light costs: our real-time renderer in a closed room



Offline: our ray tracer, CPU

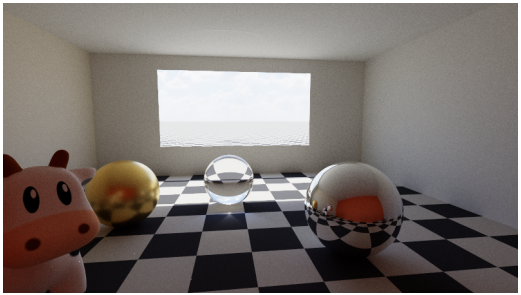
45 minutes on 12 cores

too dark: only 3 bounces

stains: the filter's guesses

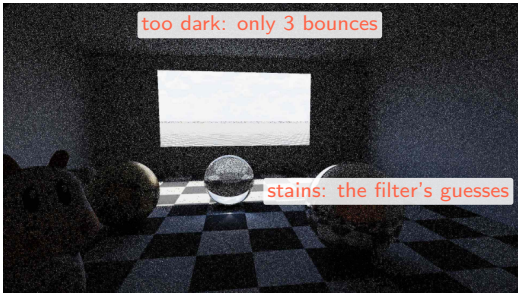
Where 16 ms breaks: light that bounces

What bounced light costs: our real-time renderer in a closed room



Offline: our ray tracer, CPU

45 minutes on 12 cores

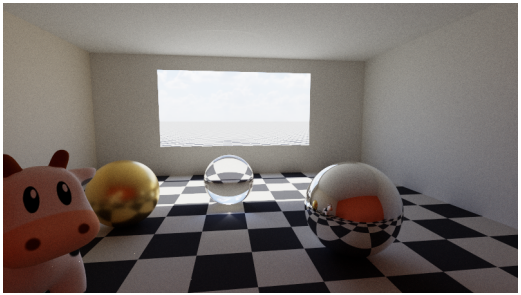


Real time: our renderer, GPU

14.9 ms a frame, RTX 5070

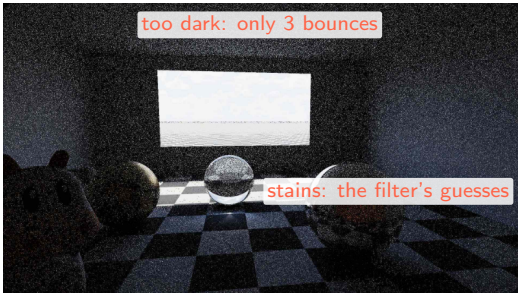
Where 16 ms breaks: light that bounces

What bounced light costs: our real-time renderer in a closed room



Offline: our ray tracer, CPU

45 minutes on 12 cores

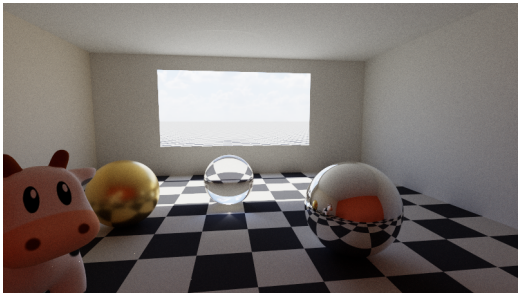


Real time: our renderer, GPU

14.9 ms a frame, RTX 5070

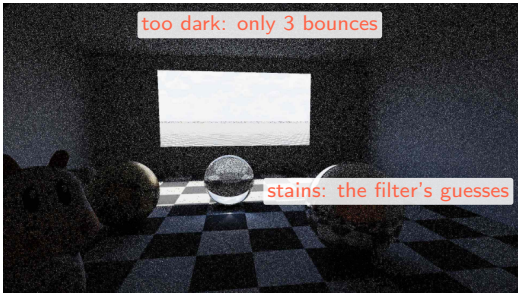
Where 16 ms breaks: light that bounces

What bounced light costs: our real-time renderer in a closed room



Offline: our ray tracer, CPU

45 minutes on 12 cores



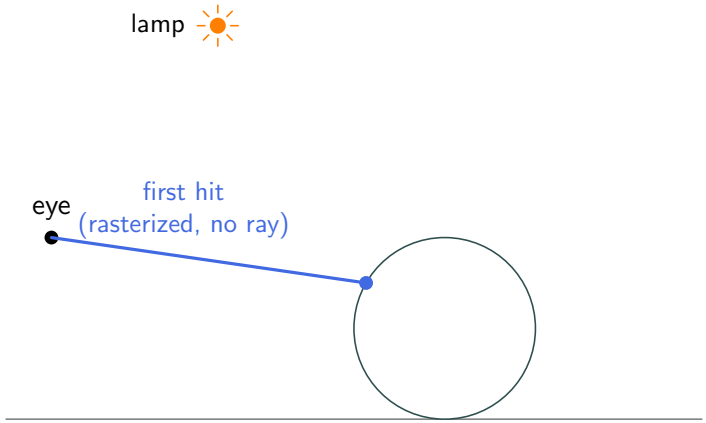
Real time: our renderer, GPU

14.9 ms a frame, RTX 5070

It is not the GPU's fault: 16 ms is too little time

Part B finds the first hit by rasterizing, and fakes the rest

So what can real time keep, and what must it fake?

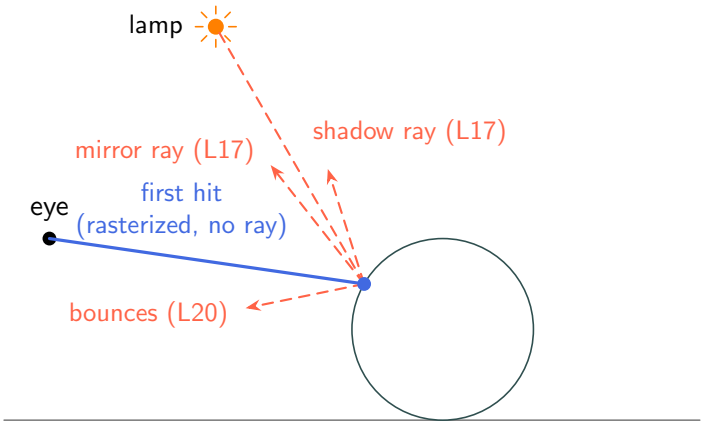


- First hit: rasterized, exact (L24)

a sketch, not to scale

Part B finds the first hit by rasterizing, and fakes the rest

So what can real time keep, and what must it fake?

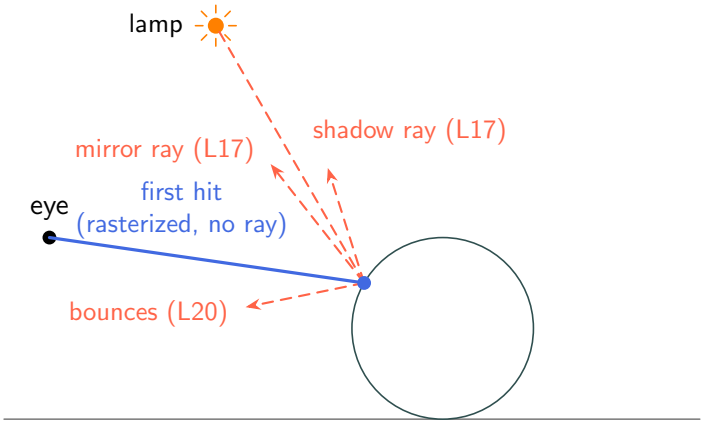


a sketch, not to scale

- First hit: rasterized, exact (L24)
- Secondary rays are too dear in 16 ms

Part B finds the first hit by rasterizing, and fakes the rest

So what can real time keep, and what must it fake?



a sketch, not to scale

- First hit: rasterized, exact (L24)
- Secondary rays are too dear in 16 ms
- Each fake leaves a visible mistake

Part B: spending the 16.7 ms

The plan for the rest of the half, as one frame

one frame, 16.7 ms



- A frame is a few passes

Part B: spending the 16.7 ms

The plan for the rest of the half, as one frame

one frame, 16.7 ms



widths: a sketch, not a measurement

- A frame is a few passes
- Each pass: lectures ahead

Part B: spending the 16.7 ms

The plan for the rest of the half, as one frame

one frame, 16.7 ms



widths: a sketch, not a measurement

each pass: one lecture, one trick, one mistake to spot

shadows: a shadow map, not the shadow ray

lighting: an environment map, not the mirror ray

effects: a screen-space guess, not the bounce

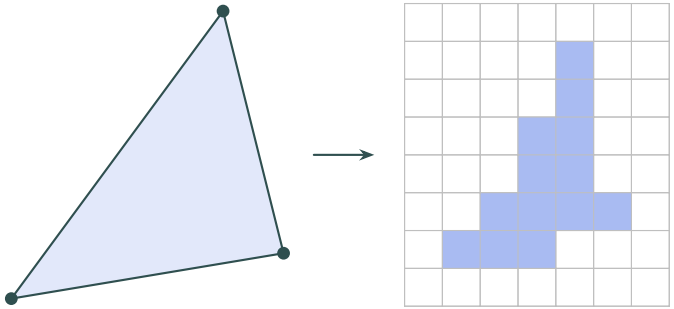
- A frame is a few passes
- Each pass: lectures ahead
- Each trick: a visible mistake

Summary: four answers, each leading to the next

1	The frame budget	16.7 ms$/2{,}073{,}600$ pixels ≈ 8 ns per pixel one core is 15 times short; no pixel waits for another
2	Building the machine	slim cores; 32 lanes per instruction; many groups a throughput machine: fast when neighbouring lanes agree
3	A real GPU	$16{,}384$ lanes, 82.6 trillion operations a second 1 TB/s of memory; circuits where the work never varies
4	Our tracer on the GPU	recursion becomes a loop; neighbours need a second pass secondary rays are incoherent, so Part B fakes them

Next lecture: one triangle's trip

Lecture 24 opens the pipeline of slide 40



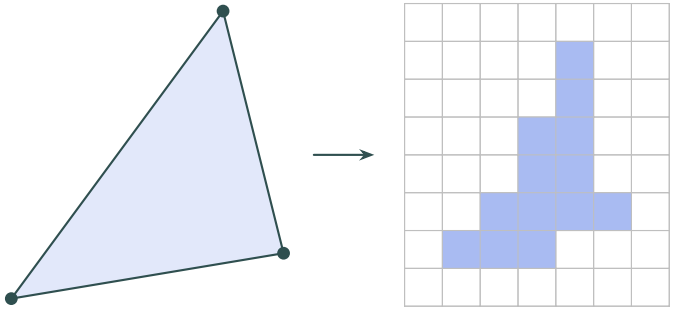
three corners

covered pixels

- The pipeline, box by box

Next lecture: one triangle's trip

Lecture 24 opens the pipeline of slide 40



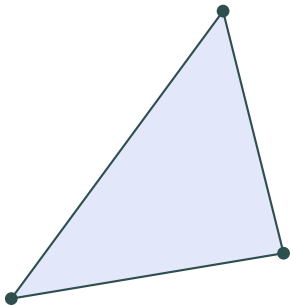
three corners

covered pixels

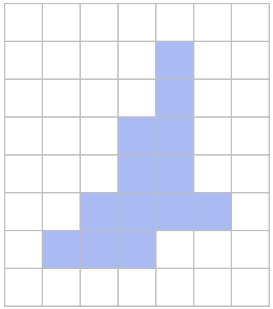
- The pipeline, box by box
- Inside the rasterizer

Next lecture: one triangle's trip

Lecture 24 opens the pipeline of slide 40



three corners



covered pixels

- The pipeline, box by box
- Inside the rasterizer
- Depth buffer against lecture 15's nearest hit

Exit Ticket

Today's question *(2 minutes)*

(a) A screen of 1280×720 pixels shows 120 pictures a second. How many nanoseconds does each pixel get?

(b) A warp of 32 lanes reaches an `if`. Four lanes take a side costing 40 ticks; the other 28 take a side costing 10. How long does the warp take?

CS 453 — Exit Ticket #23

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.