Finger Exercises 23

The GPU: Why It Exists and How It Is Built

≈ 60 minutes, plus a bonus

CS 453 — Computer Graphics

You need a pencil, a ruler and a calculator. The words are the lecture's. A *tick* is one beat of a core's clock; count one arithmetic operation per tick. A *lane* is one ALU, the yellow box that does arithmetic. The lanes of one core share one decoder, so they run the same instruction in the same tick, each on its own pixel; on NVIDIA chips 32 lanes do this together, a *warp*. A *group* is the set of pixels that one core's lanes work on together. Work is *coherent* when the lanes of a group take the same path through the program and read nearby memory. The sheet follows the lecture's four questions, one part each. Nothing here is an assignment task; the sheet is Lecture 23 only.

Short on time? Do Exercises 1, 4, 6 and 11 (about 30 minutes): the budget, a warp that splits, hiding the waits, and `trace` as a loop.

Part 1 · The frame budget

How much work is one frame?

1 Five screens, five budgets

about 6 min

Time per pixel = (1000 ms ÷ pictures a second) ÷ pixels in a picture. Then: how many ticks does that give one fast core?

A fast desktop core ticks 5 billion times a second: one tick every 0.2 ns. A VR headset has a screen for each eye, and both must be drawn for every picture.

screen	pixels	a second	ms per picture	pixels per picture	ns per pixel	ticks per pixel
old console	320 × 240	60	16.67	76,800	217	1085
phone	2400 × 1080	120	_____	_____	_____	_____
VR headset	2 × (1832 × 1920)	90	_____	_____	_____	_____
4K television	3840 × 2160	60	_____	_____	_____	_____
cinema	4096 × 2160	24	_____	_____	_____	_____

(a) Fill the table. Write each time in milliseconds first, and change it to nanoseconds only at the end (1 ms = 1,000,000 ns).

(b) One pixel of a small ray-traced scene needs about 600 operations. For which screens could one perfect core keep up?

(c) Which screen is the hardest, and which two of its numbers make it so?

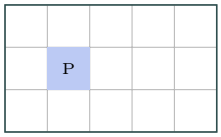
(d) A film is not drawn in real time. Suppose each frame takes 12 minutes, like the picture on the lecture's slide 2. How long would one machine need for a two-hour film at 24 frames a second? (Answer in years.) What do film studios do about it?

2 Who needs whom? about 5 min

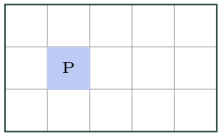
A frame splits among many workers only if no pixel needs another pixel's answer. Draw who reads whom, then sort the jobs.

Each job below makes a new picture from an old one. Sort them into three kinds: **one pass**: each pixel needs only its own old value (or the scene); **two passes**: each pixel reads other pixels of the old picture, so the old picture must be finished first; **a chain**: a pixel needs *another pixel's new value*, so the pixels must wait for each other.

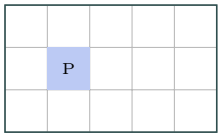
(c)



(d)



(e)



job	1 pass	2 passes	chain
(a) shade each pixel with Lambert's law	☐	☐	☐
(b) make every pixel 20 % brighter	☐	☐	☐
(c) average each pixel with its 8 neighbours	☐	☐	☐
(d) each pixel adds the new value of its left neighbour to its own	☐	☐	☐
(e) flip the picture left to right	☐	☐	☐
(f) each pixel copies its upper neighbour's new value if it is brighter	☐	☐	☐

(a) In the three small grids, draw arrows into P from every pixel that P's job reads, for jobs (c), (d) and (e). For (d), draw also the arrows into P's left neighbour. Then tick the table.

(b) Job (d) on a row of 1920 pixels: how many must finish before the last one can start? Compare with job (b).

Part 2 · Building the machine

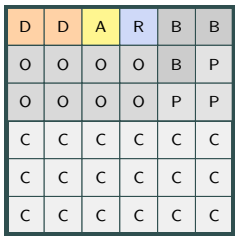
How do you build a chip with thousands of workers?

3 Count the squares

about 5 min

Where the area goes: helpers that make one job fast, against copies that make many jobs fast.

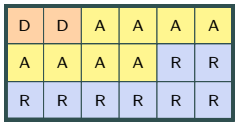
Three cores are drawn on squared paper; each square is one unit of chip area. Letters: **D** fetch and decode, **A** ALU (arithmetic), **R** registers, **O** out-of-order control, **B** branch predictor, **P** memory prefetcher, **C** cache.



a CPU core



a slim core



a SIMD core (8 lanes)

core	squares	ALUs	cores in 144 squares	ALUs in 144 squares	share that computes
CPU core	_____	_____	_____	_____	_____
slim core	4	1	36	36	$1/4 = 25\,\%$
SIMD core	_____	_____	_____	_____	_____

(a) Count the CPU core's squares of each letter, and fill its row. What fraction of the CPU core is the helpers O, B, P and C?

(b) Fill the SIMD core's row. The SIMD core saved squares in one place only. Where?

(c) Which chip finishes *one* pixel's program soonest? Which finishes a million pixels soonest? Give one reason for each.

(d) What did the SIMD chip give up to get its ALUs? (Think of an `if` that some pixels take and others do not.)

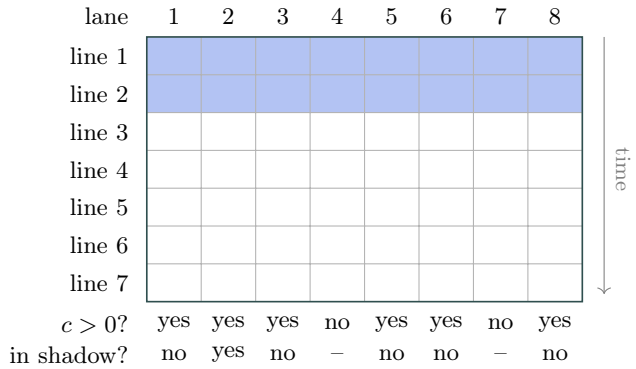
4 Be the warp

about 8 min

One decoder, eight lanes: every line that any lane needs is run for all of them, and the lanes that do not need it wait.

One core with 8 lanes shades 8 pixels with the program below. Each numbered line takes one row of time. A line runs if *at least one* lane needs it; the other lanes are masked and wait. Under the grid: does each lane's point face the lamp ($c > 0$), and is its shadow ray blocked?

```
1   c = dot(n, l)
2   if c > 0:
3       E = I * c / r2
4       if in_shadow:
5           colour = black
        else:
6           colour = k * E
    else:
7       colour = ambient
```



(a) Lines 1 and 2 are done for you: every lane works. For lines 3 to 7, shade the cell of every lane that does the work, and cross the cells of the lanes that wait.

(b) How many rows of time did the core spend? Of all its lane-rows (8× rows), what share did useful work?

(c) Lane 2 is the only lane in shadow. If its lamp were not blocked, how many rows would the core need? So what did one lane cost the other seven?

(d) How many rows if all eight lanes face the lamp and none is in shadow? And if none of them faces the lamp?

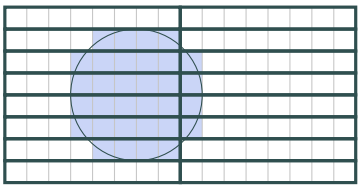
(e) Finish the rule in your own words: "A branch that no lane takes costs . . . ; a branch that one lane takes costs . . . "

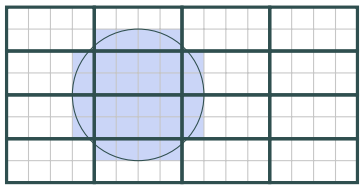
5 Which tile shape?

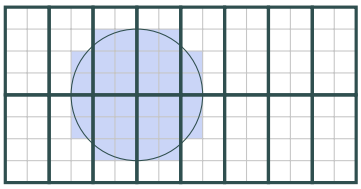
about 6 min

Neighbouring pixels usually agree, and only the tiles across an edge pay for both paths. The shape of the tile decides how many tiles an edge crosses.

The shaded pixels show a ball in a picture 16×8 pixels. The core's 8 lanes take one tile of 8 pixels at a time; the three copies use three tile shapes.


strips 8×1


tiles 4×2


tiles 2×4

(a) Circle every *split* tile: one with ball pixels and sky pixels.

(b) A tile takes 30 ticks if all ball, 5 if all sky, 35 if split (both paths). Fill the table.

(c) Which shape wins? Which would suffer most if a tall thin pole replaced the ball?

(d) GPUs cut the screen into small, nearly square blocks. Why squares?

tile	all ball	all sky	split	ticks
8×1	____	____	____	____
4×2	____	____	____	____
2×4	____	____	____	____

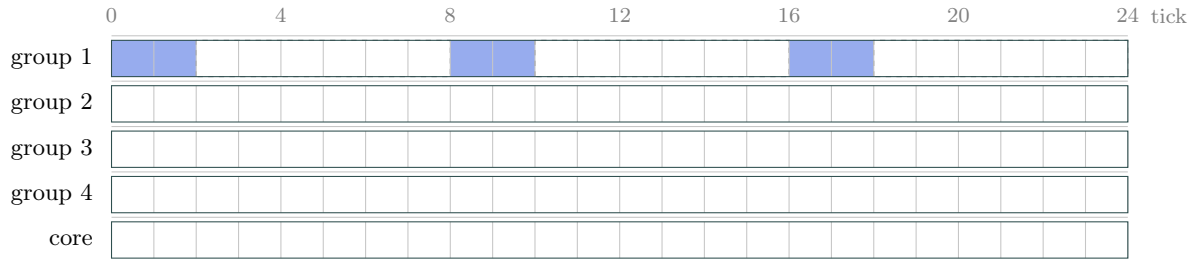
ticks: for all 16 tiles of the picture

6 Juggle the groups

about 7 min

While one group waits for memory, the core runs another. Registers decide how many groups can wait.

Every group runs the same program: it computes for 2 ticks, then reads memory, which takes 6 ticks, then computes 2 ticks, reads again, and so on. Group 1 is drawn: dark for computing, dashed for waiting. Groups 2, 3 and 4 start 2, 4 and 6 ticks later. The core can compute for only one group in each tick.



(a) With group 1 alone, in what share of the 24 ticks does the core compute?

(b) Draw groups 2, 3 and 4, and shade the core's row in every tick in which it computes for some group. Is the core ever idle once group 4 has started? Did any group finish its reads sooner?

(c) How many groups keep this core busy? Write it as a formula in the compute time C and the wait W. How many if a read takes 12 ticks?

(d) The groups wait in the registers. One scheduler of the lecture's SM has 16,384 registers and can hold at most 12 warps of 32 lanes. Fill the table.

registers per lane	32	64	96	128	255
warps that fit	12	____	____	____	____

($32 \times 32 \times 16 = 16{,}384$ would fit 16 warps, but the cap is 12.)

(e) A shader computes for 50 ticks between reads that take 350. How many warps must wait their turn to hide the reads? What is the most registers per lane the shader may use?

7 A bus or a sports car?

about 3 min

Latency is how soon one job ends; throughput is how many jobs end in a given time.

Machine A finishes a pixel in 50 ns and works on one pixel at a time. Machine B needs 2,000 ns for each pixel, but works on 4,096 pixels at once.

(a) How long does each machine take for a frame of 1280×720 pixels? Which one keeps a 60 Hz deadline?

(b) How long does each take for a single pixel, say the mouse pointer?

(c) Which machine has the lower latency, and which the higher throughput? Which of the two does a frame need, and why?

Part 3 · A real GPU

Does a real GPU look like our design?

8 Read the spec sheet

about 6 min

Lanes × ticks a second × 2 (a multiply-add) gives the arithmetic; wires × bits a second ÷ 8 gives the memory. Then share both out over the pixels, as the lecture did for the RTX 4090.

The real-time picture on the lecture's slide 2 was made on an NVIDIA RTX 5070: 48 SMs of 128 lanes, 2.51 billion ticks a second, a memory bus of 192 wires, each carrying 28 billion bits a second. A thin laptop has an integrated GPU: 8 cores of 128 lanes at 2.2 billion ticks a second, sharing the processor's memory, 89.6 GB/s.

	RTX 5070, 1920 × 1080 at 60	laptop, 1280 × 720 at 60
lanes	_____	_____
operations a second	_____	_____
operations per pixel per frame	_____	_____
memory, GB a second	_____	89.6
bytes per pixel per frame	_____	_____

(a) Fill the table.

(b) The lecture's RTX 4090 gives 82.6 trillion operations a second. How many RTX 5070s is that?

(c) Slide 2's frame took 13.5 ms on the RTX 5070. At the peak, how many operations could each of its 1920 × 1080 pixels have had? Name two reasons why the real number is smaller.

Try it tonight: your own GPU — 10 minutes, optional

Your setup check printed the name of the GPU it found. Look up that GPU on its maker's page: the number of cores (SMs, compute units or Xe-cores) and lanes in each, the clock, the memory bus and its speed (or, for an integrated GPU, the memory it shares). Add a column for it to the table above, at your own screen's size and refresh rate. Is your GPU closer to the RTX 5070 or to the laptop? Bring the number to class.

9 Lanes or circuit?

about 3 min

A job that every frame needs in the same way earns a circuit; a job that varies from one program to the next stays on the lanes.

job	lanes (your shader)	a circuit
a game's own lighting for rippling water	☐	☐
finding the pixels that a triangle covers	☐	☐
testing depth and blending a colour into the picture	☐	☐
reading a texture and blending the four nearest texels	☐	☐
testing a ray against the boxes of a BVH	☐	☐
choosing lecture 20's random bounce direction off chalk	☐	☐
decoding a video stream for playback	☐	☐
blurring a finished picture	☐	☐

(a) Tick one column in each row.

(b) The ray-box test was a program on the lanes until 2018. What changed that made it worth a circuit?

Part 4 · Our tracer on the GPU

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

10 One frame apart

about 4 min

The CPU prepares frame $N+1$ while the GPU draws frame N. The slower of the two sets the pace.

CPU, ms per frame	GPU, ms per frame	frames a second	which machine waits?
6	12	83	the CPU
14	9	_____	_____
20	20	_____	_____
4	18	_____	_____

(a) Fill the table.

(b) For each of the three rows, which machine would you make faster to reach 60 frames a second? Would a faster GPU help the second row?

(c) In the worked row, a key is pressed just as the CPU begins a frame. At the earliest, how many ms later is that frame on the screen?

11 Unroll the recursion

about 7 min

A shader may not call itself. The recursion's multiplications become a running weight.

Our path tracer, chalk and glowing surfaces only, with the sky's colour `sky`. Its colours are grey here, so each is one number.

recursive (lecture 20)

```
def trace(ray, depth):
    if depth == 0:
        return 0
    rec = scene.hit(ray)
    if rec.t == math.inf:
        return sky
    material = rec.surface.material
    if isinstance(material, DiffuseLight):
        return material.emission
    x = ray.evaluate(rec.t)
    d = rec.normal + random_unit_vector()
    return material.albedo * \
           trace(Ray(x, d), depth - 1)
```

as a loop (lecture 23)

```
def trace(ray, depth):
    weight = 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 0
```

One path from the eye meets chalk of albedo 0.8, then chalk of 0.5, then chalk of 0.9, and then leaves the scene: the sky, `sky` $= 2.0$.

(a) Run the recursive version with `depth = 4`. Draw the calls as a stack of boxes, one per call, each with the number it is waiting to multiply by. What does the first call return?

(b) Run the loop: write `weight` after each pass of the loop, and what it returns. Same answer?

(c) Run both with `depth = 3`. Do they still agree?

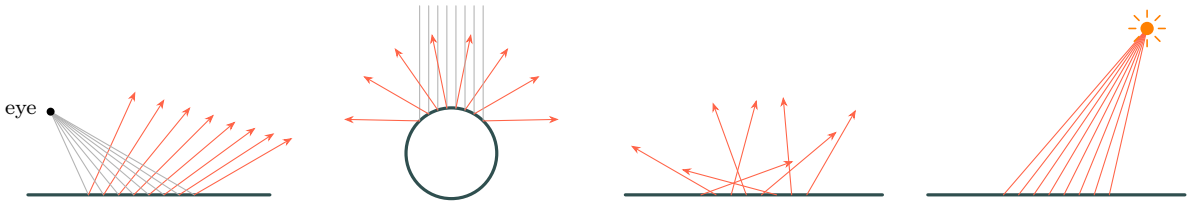
(d) A mirror, `Reflective`, keeps its colour in `material.albedo` too, and sends the ray on along `ray.direction.reflect(rec.normal)`. Write the three lines the loop needs for it, before the chalk lines: `if isinstance(material, Reflective):`, then two lines and `continue`.

(e) Why the GPU forbids it: each waiting call needs about 256 bytes. A path 12 bounces deep, in every lane of an RTX 5070 (48 SMs, each holding 48 warps of 32 lanes): how many MB of stack? Compare it with all the registers on the chip (48 SMs × 65,536 registers of 4 bytes). How much does the loop keep instead?

12 Coherent or scattered? about 5 min

Eight neighbouring lanes, eight rays. If the rays stay together, the lanes agree; if they scatter, the lanes hit different objects, take different branches and read memory far apart.

Each picture shows the second rays of eight neighbouring pixels.



W: a flat mirror X: a mirror ball Y: chalk, random bounces Z: shadow rays to a lamp

(a) Label each picture *coherent* or *scattered*.

(b) In picture W, extend the red rays backwards below the mirror with your ruler. They meet at one point. What is that point, and why does it make the mirror's rays as cheap as first rays?

(c) In picture X, the rays stay together when the ball is large and spread when it is small. Why? (Compare the normals at two neighbouring hit points.)

(d) Lecture 21's lamp has a size: each shadow ray aims at a random point on it. Is that more like Z or like Y? And is it the ray's being *secondary* that makes it dear, or something else?

13 Bonus: the longest path wins ★ about 10 min

A warp is done only when its last lane is done. With paths of random length, most lanes wait for the longest one.

Lecture 20's paths can stop at random: after each bounce, a path goes on with probability 1/2 and stops otherwise. So a path has $L = 1$ bounce with probability 1/2, $L = 2$ with 1/4, $L = 3$ with 1/8, and so on. Each bounce takes the same time. The 32 lanes of a warp trace 32 paths side by side, and the warp keeps going until its longest path stops.

(a) Show that the probability that one path is *longer* than k bounces is $(1/2)^k$, and that the mean length is 2. (The sum $\sum_{k\geq 0} P(L > k)$ gives the mean.)

(b) The probability that *all* 32 paths stop within k bounces is $(1 - 2^{-k})^{32}$. Fill the table.

k	3	4	5	6	7	8
$P(\text{longest} \leq k)$	0.014	____	____	____	____	0.882

(c) The expected longest path is $\sum_{k\geq 0}\big(1 - (1 - 2^{-k})^{32}\big)$. Add the terms until they are below 0.01. About how many bounces does the warp run?

(d) While the warp runs, what share of its lane-bounces does useful work? What would it be for a core with 8 lanes? With 1?

(e) What does this say about tracing many bounces on a GPU, and about Part B's plan to skip them?

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides.

1a.

screen	ms per picture	pixels per picture	ns per pixel	ticks per pixel
phone	$1000/120 = 8.33$	2,592,000	3.2	16
VR headset	$1000/90 = 11.11$	$2 \times 3{,}517{,}440 = 7{,}034{,}880$	1.6	8
4K television	16.67	8,294,400	2.0	10
cinema	$1000/24 = 41.67$	8,847,360	4.7	24

For example the phone: 8.33 ms = 8,333,333 ns, shared by 2,592,000 pixels, is 3.2 ns; at 0.2 ns a tick, 16 ticks.

1b. Only the old console: 1085 ticks for 600 operations. Every modern screen gives a core between 8 and 24 ticks.

1c. The VR headset: it has the most pixels to draw (two screens) *and* a short time (90 pictures a second). It gets 1.6 ns, eight ticks, per pixel.

1d. $2 \times 3600 \times 24 = 172{,}800$ frames; times 12 minutes is 2,073,600 minutes = 34,560 hours, about 3.9 years. Studios use render farms: thousands of machines, each drawing its own frames. The frames are independent, like the pixels.

2a. (c): arrows into P from its 8 neighbours. (d): one arrow into P from its left neighbour, and one into that neighbour from *its* left neighbour, and so on to the row's start. (e): one arrow into P from its mirror pixel, the same distance from the other edge (P is in the second column, so its mirror is in the second column from the right).
(a) one pass (b) one pass (c) two passes (d) a chain (e) two passes (f) a chain. In (c) and (e) the old picture is read, so it must be finished, but no new value is waited for. In (d) and (f) each pixel waits for a neighbour's *new* value.

2b. All 1919 others: they run one after another. In (b) none waits: all 1920 can run at once. (Clever GPU programs do (d) in about $\log_2 1920 \approx 11$ rounds instead, by adding pairs, then pairs of pairs; but not as written.)

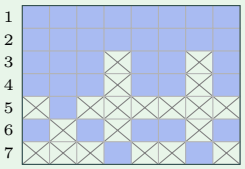
3a. CPU core: D 2, A 1, R 1, O 8, B 3, P 3, C 18: 36 squares, 1 ALU. In a chip of 144: 4 cores, 4 ALUs; $1/36 \approx 2.8\,\%$ computes. The helpers are 32/36, about 89 %.

3b. SIMD core: D 2, A 8, R 8: 18 squares, 8 ALUs. 8 cores, 64 ALUs; $8/18 \approx 44\,\%$. It saved only on the decoder: eight slim cores would need 16 squares of D, the SIMD core needs 2.

3c. One pixel: the CPU chip, because its helpers let one job go fast (no waiting on branches, memory close in the cache). A million pixels: the SIMD chip, because 64 ALUs work at once against 4.

3d. Freedom: its 8 lanes must run the same instruction. An `if` that some pixels take and others do not runs both sides.

4a.



Line 3 and line 4: lanes 1, 2, 3, 5, 6, 8 (they face the lamp). Line 5: lane 2 only. Line 6: lanes 1, 3, 5, 6, 8. Line 7: lanes 4 and 7.

4b. 7 rows. Useful: $8+8+6+6+1+5+2 = 36$ lane-rows of 56: 64 %.

4c. 6 rows (line 5 is skipped: no lane needs it). Lane 2 made all eight lanes spend a row, seven of them doing nothing.

4d. All facing, none in shadow: lines 1, 2, 3, 4, 6, so 5 rows. None facing: lines 1, 2, 7, so 3 rows.

4e. A branch that no lane takes costs nothing; a branch that one lane takes costs as much as if every lane took it.

5a, b.

tile shape	all ball	all sky	split	ticks
8×1	0	6	10	$10 \times 35 + 6 \times 5 = 380$
4×2	2	8	6	$6 \times 35 + 2 \times 30 + 8 \times 5 = 310$
2×4	0	8	8	$8 \times 35 + 8 \times 5 = 320$

The ball's pixels, counting rows and columns from 1 at the bottom left: rows 3 to 6 are columns 4 to 9; rows 2 and 7 are columns 5 to 8.

5c. 4×2 wins. A tall thin pole crosses every row, so it splits a strip in every row it crosses: the 8×1 strips suffer most. (A wide low wall would hurt the 2×4 tiles instead.)

5d. An edge can run in any direction. For a given number of pixels, a square tile has the shortest border, so on average an edge crosses the fewest square tiles. Squares are the best guess when nothing is known.

6a. 6 of 24 ticks: 25 %, that is $C/(C+W) = 2/8$.

6b. Group g computes in ticks $2(g-1)$ to $2(g-1)+2$, then every 8 ticks after. The four groups take turns: $1, 2, 3, 4, 1, 2, \ldots$, and the core's row is full from tick 0 to tick 24. No group finished sooner: each still waits 6 ticks for every read. The wait is hidden, not shortened.

6c. $(C+W)/C = 1 + W/C = 1 + 6/2 = 4$ groups. With $W = 12$: $1 + 12/2 = 7$ groups.

6d. A warp needs $32 \times r$ registers. 64: 16,384/2,048 = 8 warps; 96: 16,384/3,072 = 5.3, so 5; 128: 4; 255: 16,384/8,160 = 2.0, so 2.

6e. $1 + 350/50 = 8$ warps. The table says 64 registers per lane still fit 8; at 96 only 5 fit, and the reads show.

7a. 921,600 pixels. A: 921,600 × 50 ns = 46.1 ms, too slow for 16.7 ms. B: 921,600/4,096 = 225 rounds of 2,000 ns = 0.45 ms. B keeps the deadline easily.

7b. A: 50 ns. B: 2,000 ns, forty times longer.

7c. A has the lower latency, B the higher throughput. A frame needs throughput: it is done only when its last pixel is done, and nobody waits for the first.

8a.

	RTX 5070	laptop
lanes	$48 \times 128 = 6{,}144$	$8 \times 128 = 1{,}024$
operations a second	$6{,}144 \times 2.51 \times 10^9 \times 2 = 30.8$ trillion	$1{,}024 \times 2.2 \times 10^9 \times 2 = 4.5$ trillion
per pixel per frame	$30.8 \times 10^{12}/(60 \times 2{,}073{,}600) \approx 248{,}000$	$4.5 \times 10^{12}/(60 \times 921{,}600) \approx 81{,}500$
memory	$192 \times 28/8 = 672$ GB/s	89.6 GB/s
bytes per pixel per frame	$672 \times 10^9/124{,}416{,}000 \approx 5{,}400$	$89.6 \times 10^9/55{,}296{,}000 \approx 1{,}620$

8b. $82.6/30.8 \approx 2.7$ RTX 5070s.

8c. $30.8 \times 10^{12} \times 0.0135/2{,}073{,}600 \approx 200{,}000$ operations per pixel. Less in practice: split branches leave lanes masked, memory waits are not always hidden (too few warps, or the bytes run out first), and some of the time goes to work that is not pixels.

9a. Lanes: the water's lighting, the chalk bounce, the blur (each game or effect wants its own). Circuits: the covered pixels (the rasterizer), depth and blending, texture filtering (the texture units), the ray-box tests (the RT core), video decoding (a video unit).

9b. Games began to trace rays in every frame, so the same test was needed billions of times a second, always done the same way. A job that every frame needs eventually earns a circuit.

10a. $(14, 9)$: $1000/14 \approx 71$, the GPU waits. $(20, 20)$: 50, neither waits. $(4, 18)$: $1000/18 \approx 56$, the CPU waits. The pace is set by the slower machine: 1000/ max.

10b. Row 2: the CPU (to 16.7 ms); a faster GPU would only wait longer. Row 3: both. Row 4: the GPU.

10c. At least $6 + 12 = 18$ ms: the CPU prepares it, then the GPU draws it, and only then can the screen show it (often one refresh later still).

11a. Four calls: depth 4 (waits to multiply by 0.8), depth 3 (×0.5), depth 2 (×0.9), depth 1 (the sky: returns 2.0). Then back up: $0.9 \times 2.0 = 1.8$, $0.5 \times 1.8 = 0.9$, $0.8 \times 0.9 = 0.72$. The first call returns 0.72. (Four calls are on the stack at the deepest point.)

11b. `weight`: 1, then 0.8, 0.4, 0.36; the fourth pass meets the sky and returns $0.36 \times 2.0 = 0.72$. The same: the loop does the multiplications on the way down instead of on the way back.

11c. Yes: both give 0. The recursion reaches `depth == 0` after three chalk hits; the loop runs out of passes after three and returns 0. (The path is cut off before it finds the sky.)

11d.

```
if isinstance(material, Reflective):
   weight = weight * material.albedo
   ray = Ray(x, ray.direction.reflect(rec.normal))
   continue
```

(after `x` is computed, before the chalk lines).

11e. $48 \times 48 \times 32 = 73{,}728$ lanes; $\times 12 \times 256$ bytes = 226 MB. The registers of the whole chip: $48 \times 65{,}536 \times 4$ bytes = 12.6 MB, eighteen times less. The loop keeps one weight (three numbers for a colour) and one ray.

12a. W coherent, X scattered (spread), Y scattered, Z coherent.

12b. The mirror image of the eye, as far below the mirror as the eye is above it. The reflected rays are the first rays of a second camera at that point: neighbours stay neighbours, hit the same objects and read nearby memory.

12c. On a large ball, two neighbouring hit points have nearly the same normal, so their rays turn by nearly the same angle. On a small ball the normal changes fast from one point to the next, and so does the reflected ray.

12d. Between the two: all rays aim at one lamp, but at random points on it, so neighbours may differ in what they hit near the lamp's edge. What makes a ray dear is randomness and curvature, not being secondary: Z and W are secondary rays and still coherent.

13a. A path is longer than k only if it went on after each of its first k bounces: $(1/2)^k$. Mean $= \sum_{k \geq 0} 2^{-k} = 1 + 1/2 + 1/4 + \cdots = 2$.

13b. $k = 4$: 0.127; 5: 0.362; 6: 0.604; 7: 0.778.

13c. Terms for $k = 0, 1, 2, \ldots$: $1, 1.000, 1.000, 0.986, 0.873, 0.638, 0.396, 0.222, 0.118, 0.061, 0.031, 0.016, 0.008, \ldots$; the sum is about 6.4 bounces.

13d. $2/6.4 \approx 31\,\%$ of the lane-bounces are useful. With 8 lanes the longest of 8 is about 4.4: 45 %. With 1 lane:

100 % (a CPU core).

13e. The more bounces and the more random their number, the more of a warp's time is spent waiting for its last lane: about two thirds here, before any split branch or scattered read is counted. Part B's tricks avoid the bounces altogether; lecture 40 shows how far the hardware has come in doing them anyway.