Advanced Topics

Presentation by Kate Baker, Lvmin Zhang, and Connor Ding

Advanced Topics: Lvmin Zhang

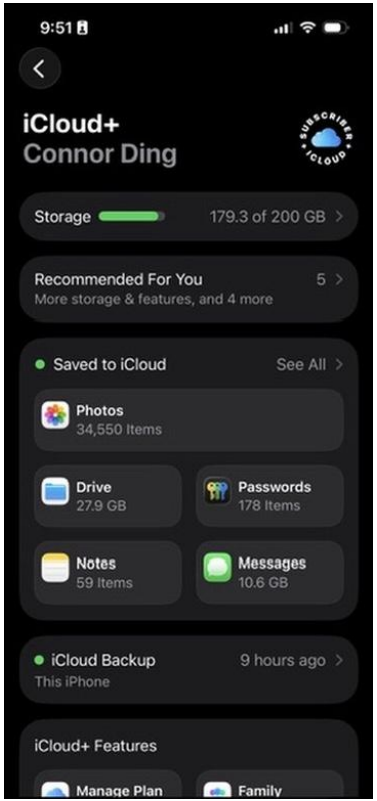
Presentation by Kate Baker, Lvmin Zhang, and Connor Ding

Advanced Topics: Connor Ding

Introduction to Compression

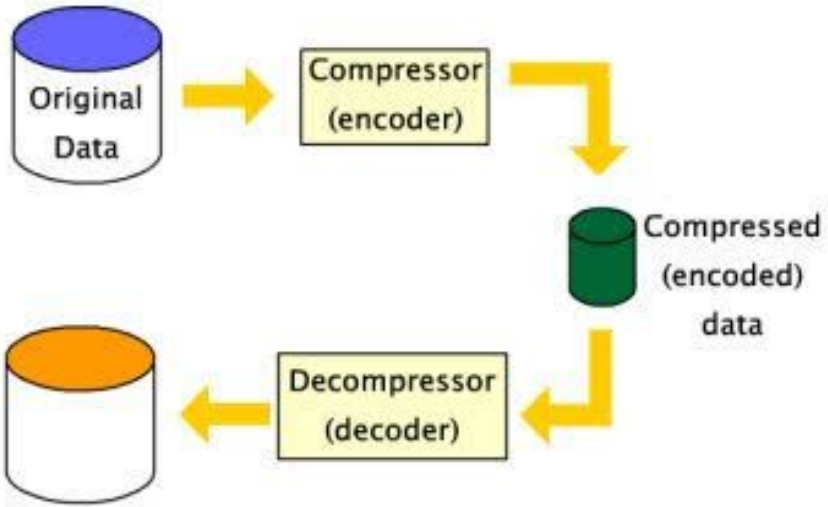
Presentation by Kate Baker, Lvmin Zhang, and Connor Ding

Constraints: Storage and Network Bandwidth



Solution – Compression!

- Goal: reduce the size of the data for efficient storage and communication
- General Pipeline
  - Encode data into bits
  - Store/transmit data depending on the use cases
  - Decode the stored data into its original form



Example: Communicating Your Emotion

Your Task: Every minute, you will communicate with your friend about how you feel.

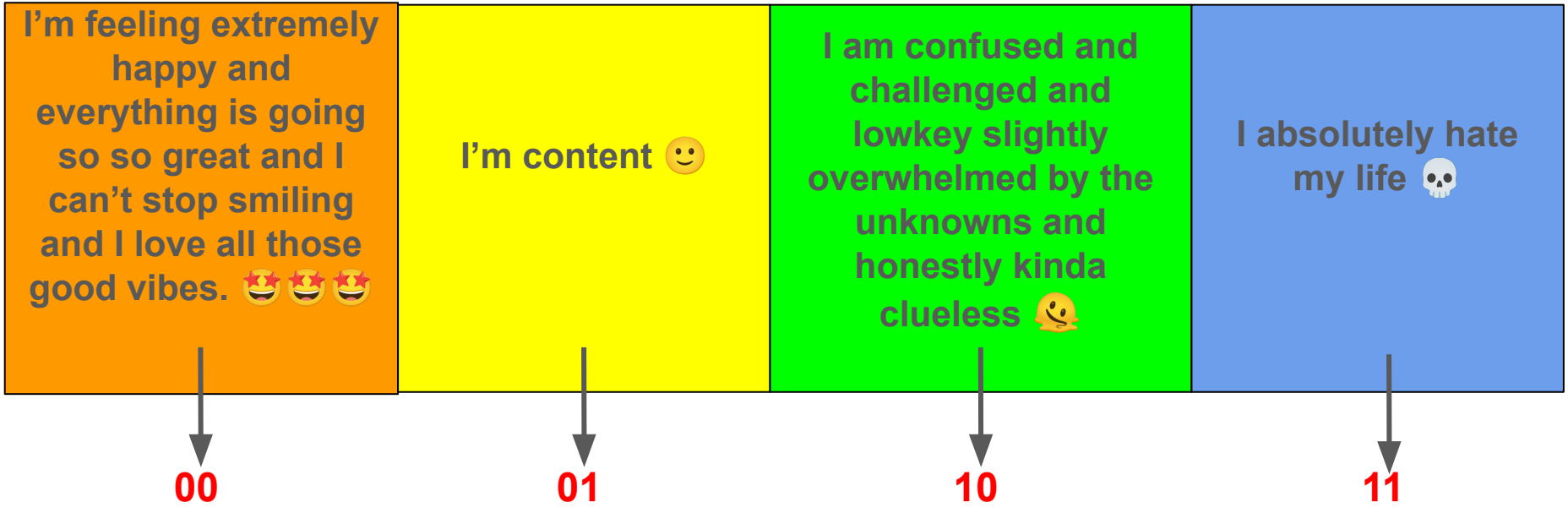
Example: Communicating Your Emotion

I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless. 🫠	I absolutely hate my life 💀

Example: Communicating Your Emotion

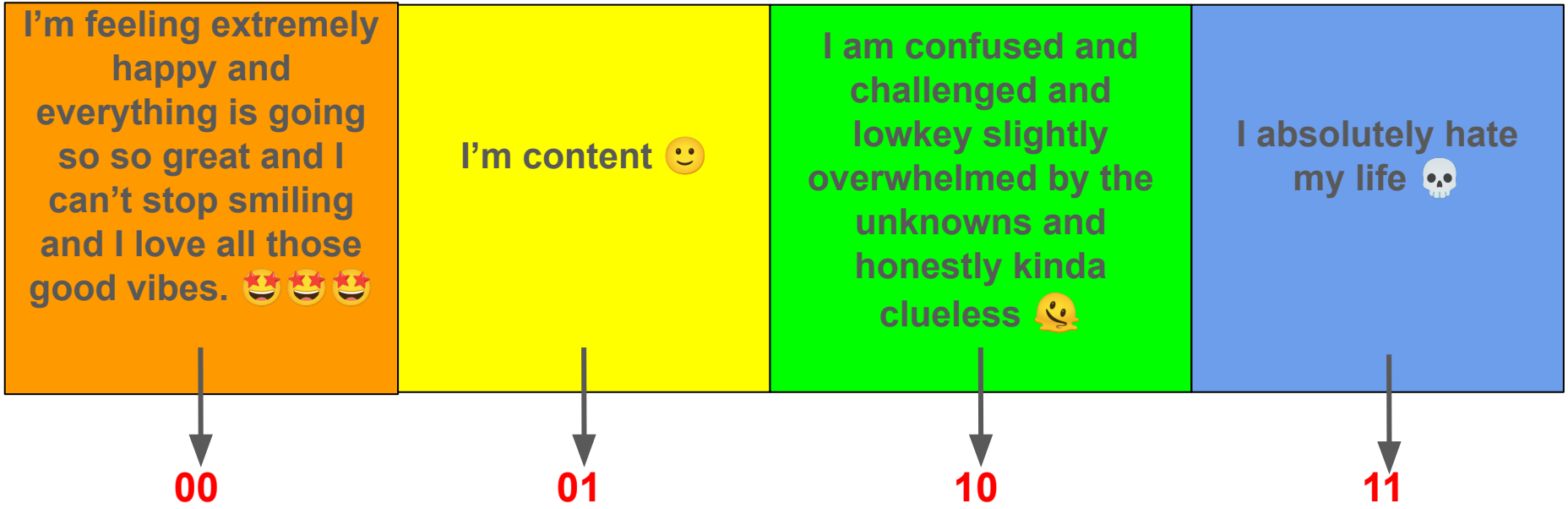
I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless 🫠	I absolutely hate my life 💀
00	01	10	11

Example: Communicating Your Emotion



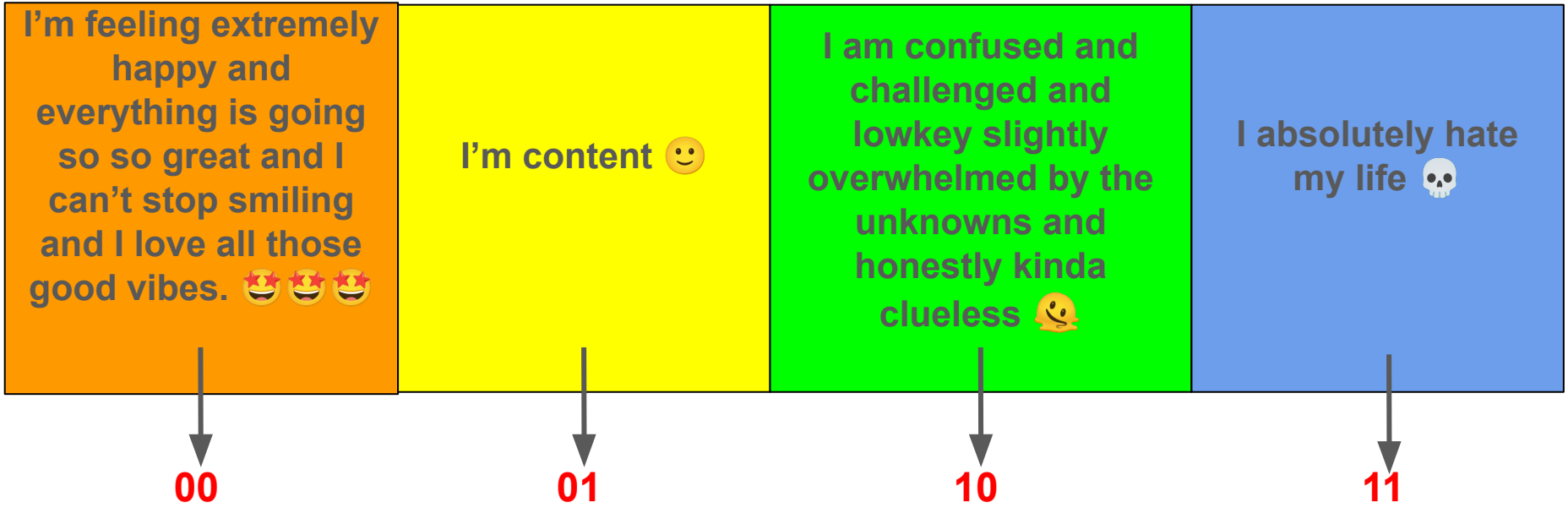
- In an eight-minute span, your friend will receive something like this: **0101010101010101**
- What does this message (or *bitstream)* translate to?

Example: Communicating Your Emotion



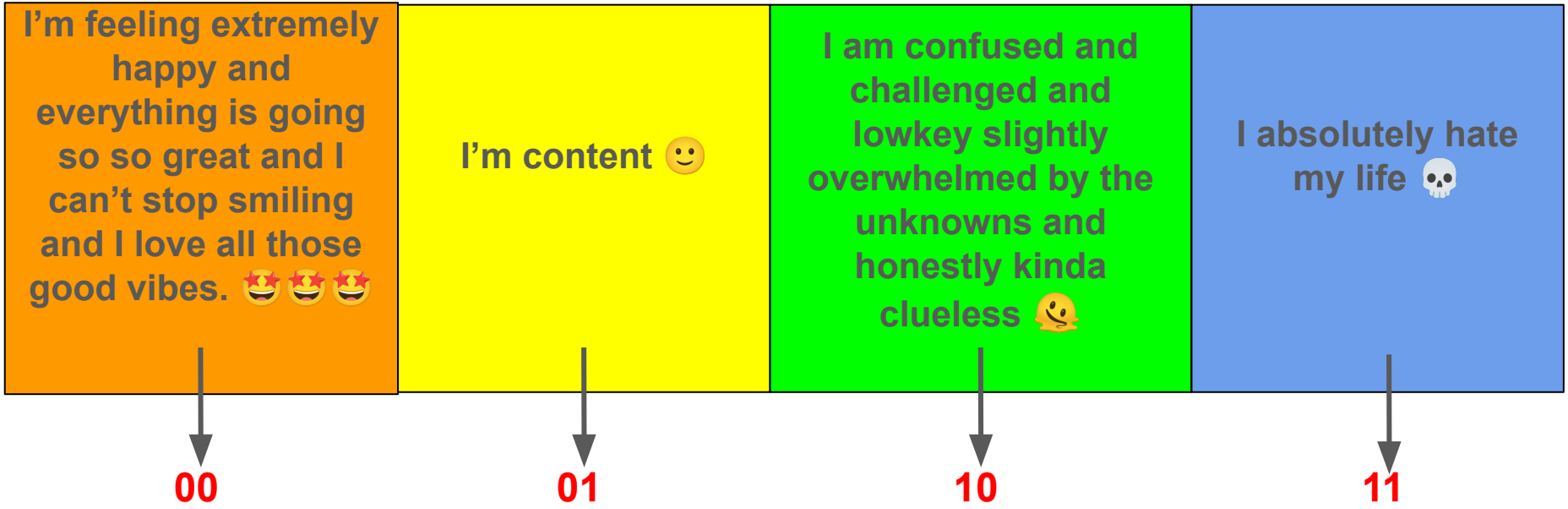
- In an eight-minute span, your friend will receive something like this: **0101010101010101**
- What does this message (or *bitstream)* translate to? → **"I'm content x8"**

Example: Communicating Your Emotion



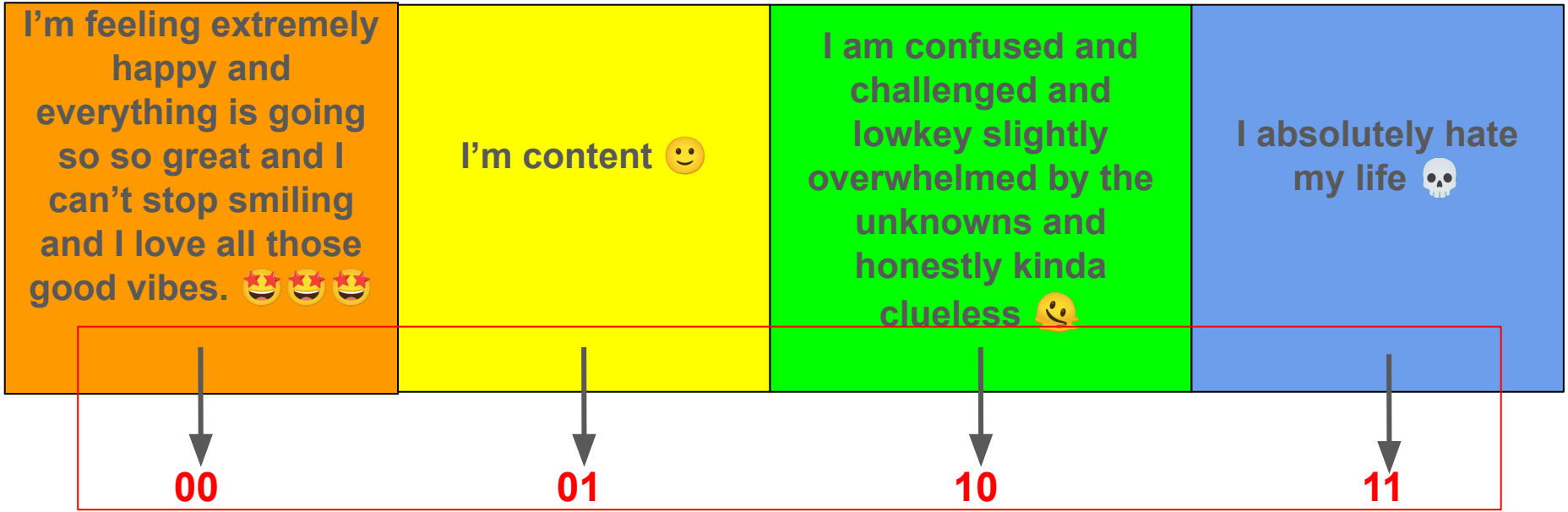
- For the bitstream **0101010101010101**, how much compression was achieved?

Example: Communicating Your Emotion



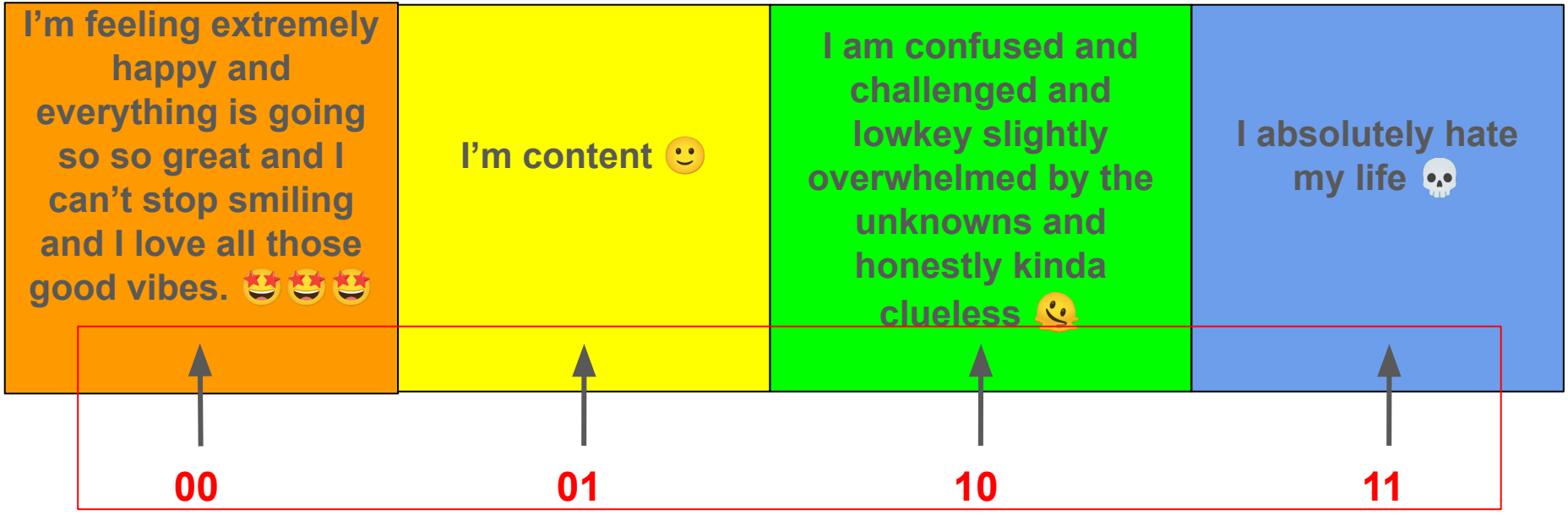
- For the bitstream **0101010101010101**, how much compression was achieved?
- **2 bits for 12 characters → 6x**

Example: Communicating Your Emotion



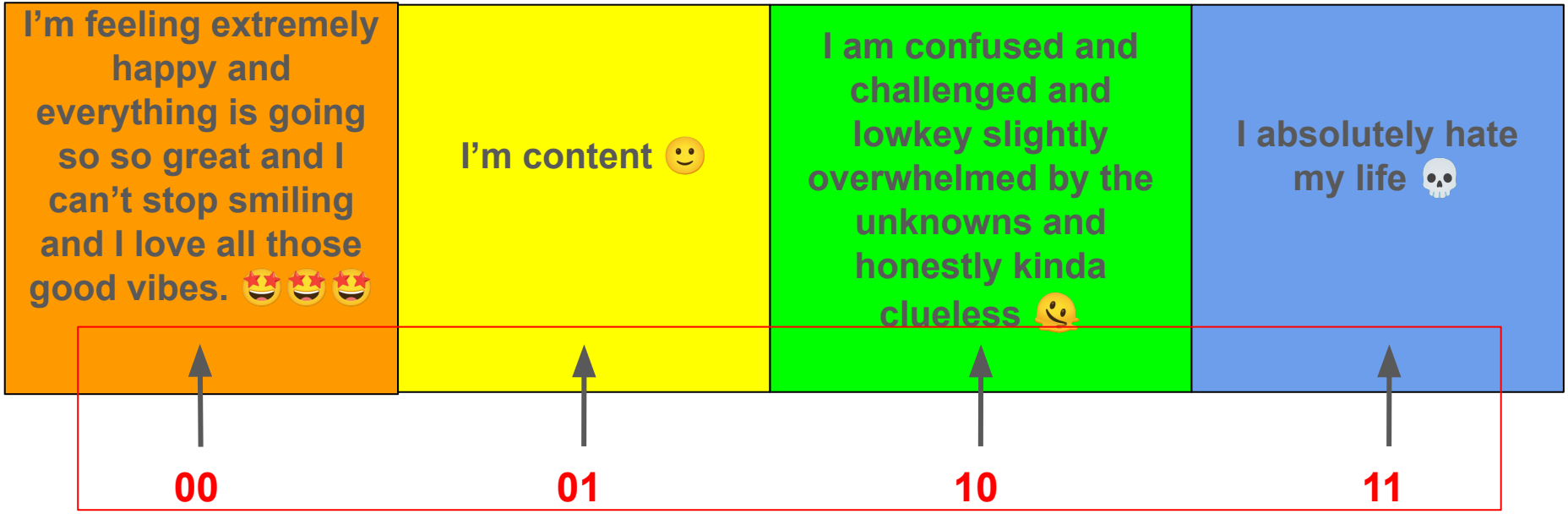
Encoder/Compressor

Example: Communicating Your Emotion



Decoder/Decompressor

Example: Communicating Your Emotion



Decoder/Decompressor

- **0101010101010101 → 01|01|01|01|01|01|01|01**

Example: Uneven Distribution of Emotion

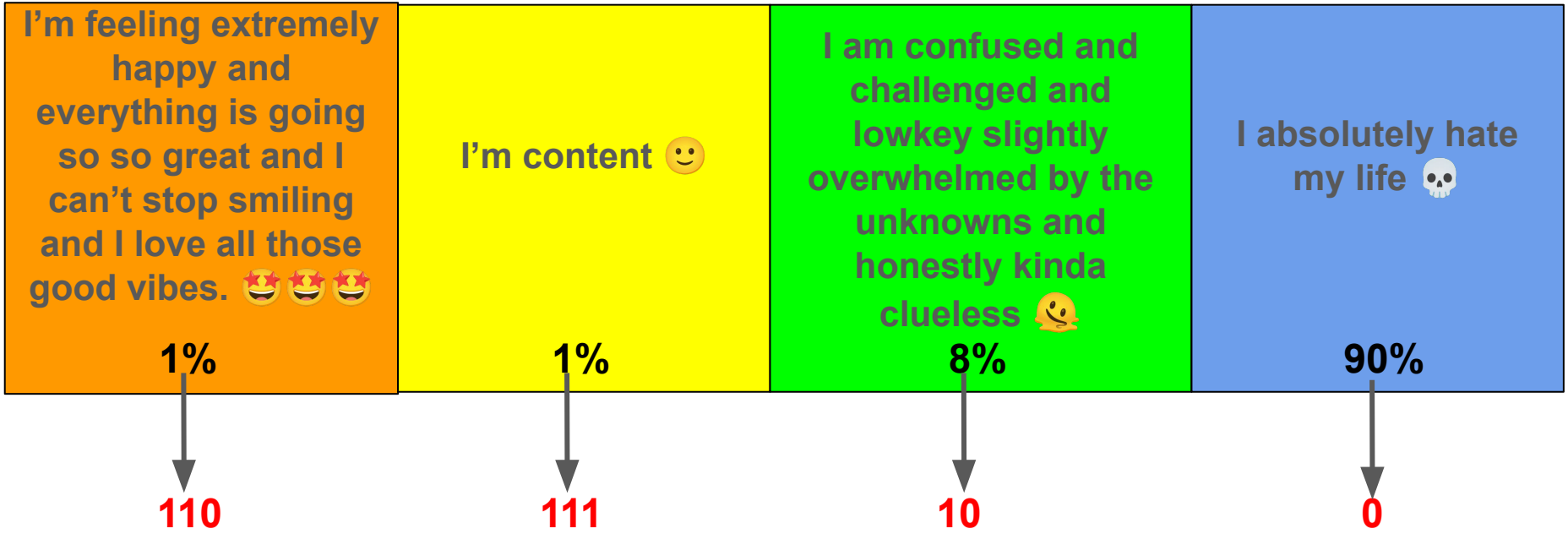
I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless 🫠	I absolutely hate my life 💀
1%	1%	8%	90%

- You know that you'll tell your friend that you hate your life the whole time
- Can you modify the encoder to achieve better compression?
- **Hint: Assigning less bits to emotion w/ higher probability**!

Example: Uneven Distribution of Emotion

I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless 🫠	I absolutely hate my life 💀
1%	1%	8%	90%
110	111	10	0

Example: Uneven Distribution of Emotion



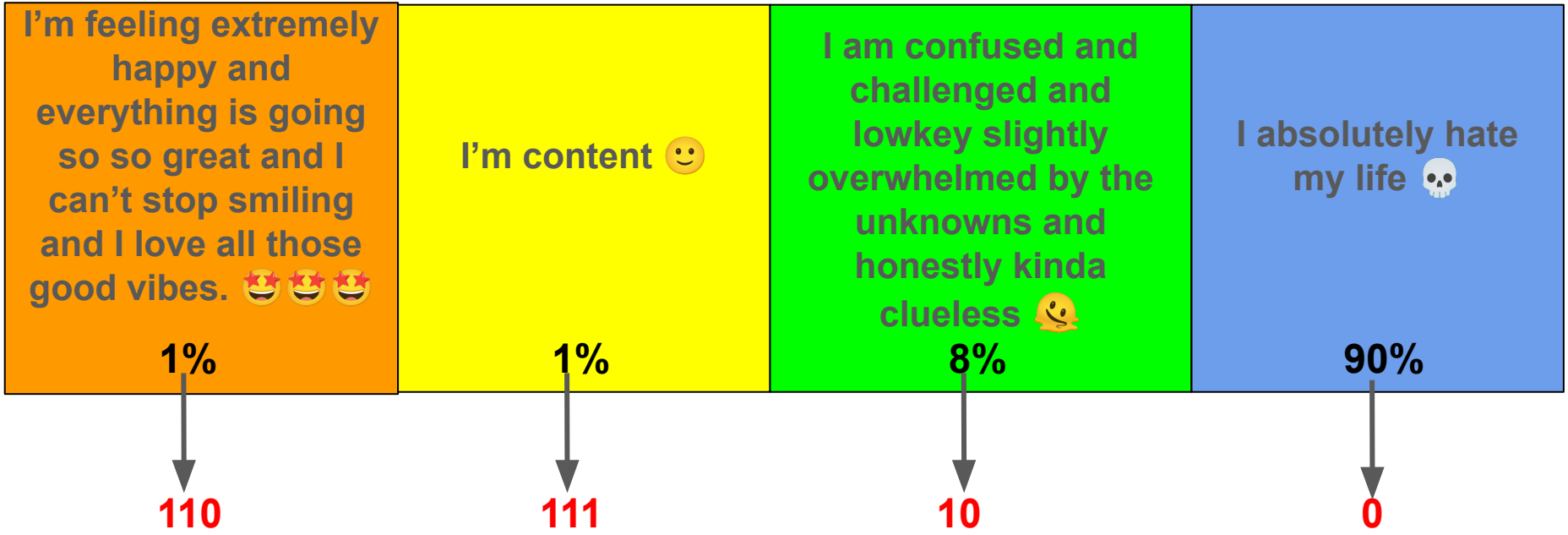
- A very probable 10-minutes span of emotions: 💀💀💀💀💀🫠💀💀🤩💀
- Encode: **0000010001100**
- **13 bits → 1.3 bits/emotion**

Example: Uneven Distribution of Emotion

I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless 🫠	I absolutely hate my life 💀
1%	1%	8%	90%
00	01	10	11

- A very probable 10-minutes span of emotions: 💀💀💀💀💀🫠💀💀🤩💀
- Encode with the previous 2 bits/emotion scheme: **11111111111011110011**
- **20 bits → 7 bits more than the previous scheme!**

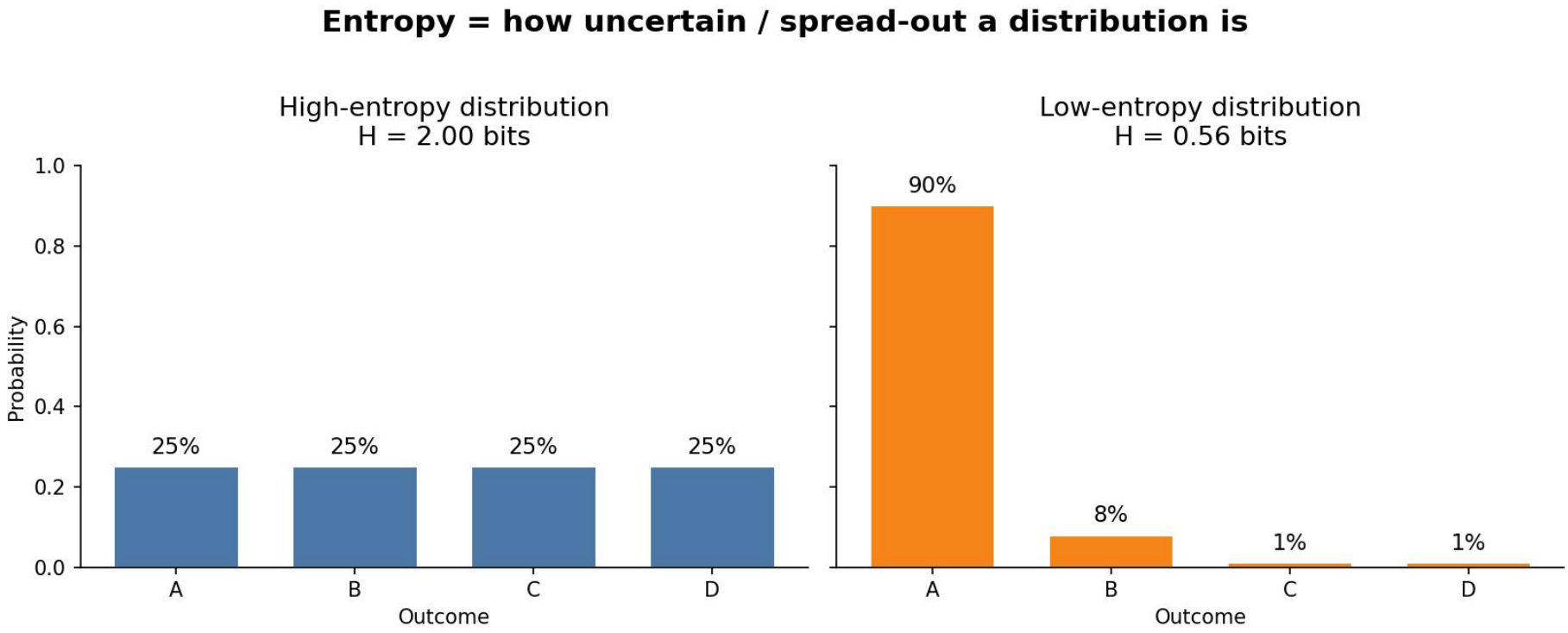
Example: Uneven Distribution of Emotion



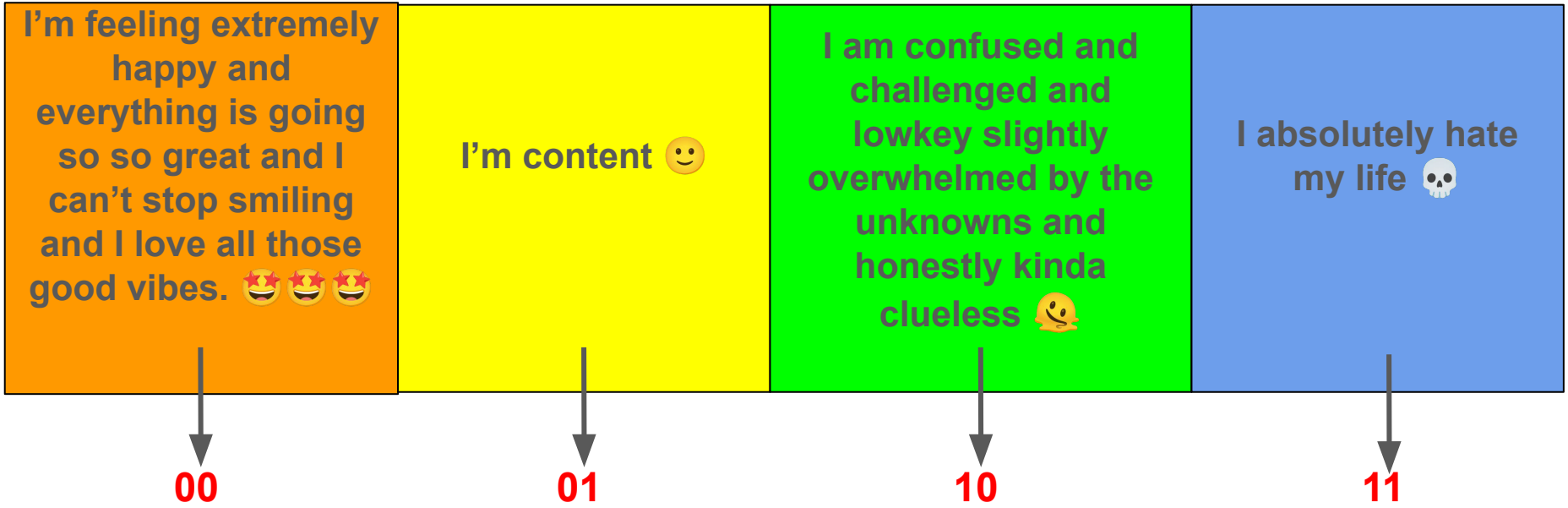
- This distribution has a lower **entropy**
  - Lower entropy ⇒ Less uncertainty in a probability distribution ⇒ Less bits to represent data ⇒ better compression!
  -

Entropy: Measure of Compressibility

$$H = -\sum p(x)\log p(x)$$



Example: Communicating Your Emotion



Your friend just received the following message sequence from you: **0000000000**

- For the past five minutes, you've been feeling great 🤩!
- This is an example of **lossless compression**

Example: Communicating Your Emotion

What if you only want to communicate the four emotions as binaries?

- New coding scheme: **0 → Good, 1 → Bad**
- Now you can communicate with **even less bits → better compression!**

Example: Communicating Your Emotion

- However, your friend won't be able to decode exactly how you feel.
  - Hence the scheme is **lossy**.

Example: Communicating Your Emotion

Consider the following message coded with the lossy scheme: **001100000**

- 2x more compression compared to a 2 bits/emotion lossless scheme
- But your friend can't decode your emotion exactly

Example: Communicating Your Emotion

Note that lossy compression has a lossless compression component!

- We map four emotions down to two (🤩🙂 → **good = 0;** 🫠💀 → **bad = 1**)
  - A process known as **quantization**.
- Then we code those two emotions **losslessly** into bits.

Example: Communicating Your Emotion

Two important measurements of lossy compression

- **Rate (R):** Number of bits needed to code source data (usually measured by bits/unit source, also applies to lossless compression)
- **Distortion (D):** The difference between the original source data and the recovered source data from the compressed bitstream

For You to Think About…

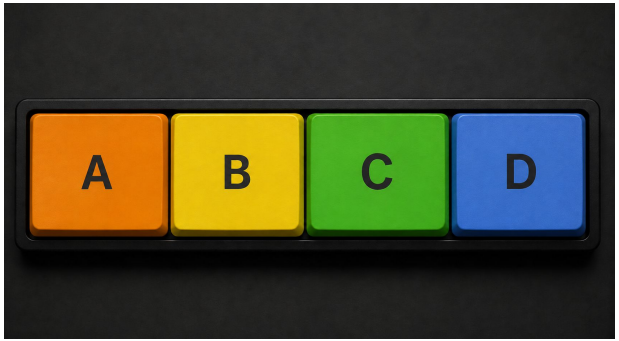
I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless 🫠	I absolutely hate my life 💀
14%	70%	15%	1%

For you to think about on your own…

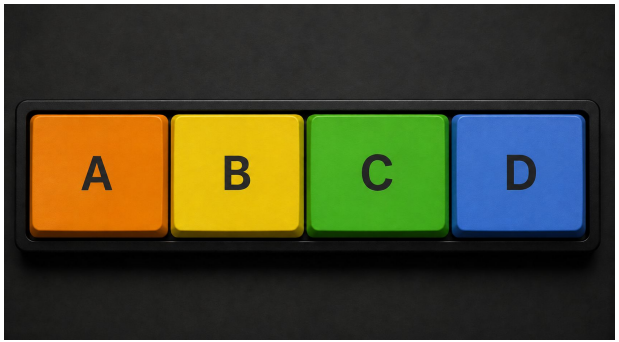
- What is the entropy of this distribution? Is it more (losslessly) compressible than a uniform distribution? Is it more than the extremely pessimistic case?
- Can you design a lossless compressor? A lossy one? What is the rate?
- For the lossy compressor, how do you quantify the difference between the reconstructed emotion and the original emotion?

Why Bits?

I'm feeling extremely happy and everything is going so so great and I can't stop smiling and I love all those good vibes. 🤩🤩🤩	I'm content 🙂	I am confused and challenged and lowkey slightly overwhelmed by the unknowns and honestly kinda clueless. 🫠	I absolutely hate my life 💀



Example: Why Bits?



- Bits are the simplest universal representation of information source
- Binary states are the easiest and the most noise-free way for hardwares to represent/communicate
- Great mathematical properties
  - Boolean algebra
  - Information theory

Lossy Image Compression

Reference image · RGB · 5000 × 3333 = 16,665,000 pixels

Red channel
5000×3333 × 8 bits

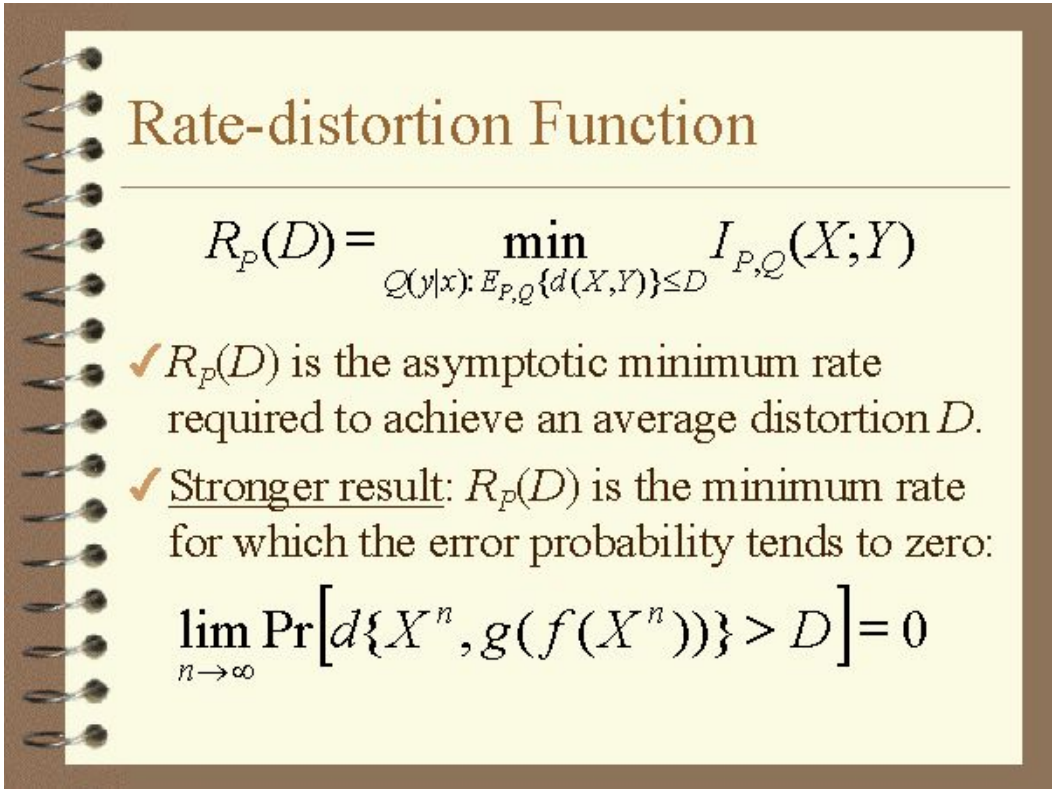
Green channel
5000×3333 × 8 bits

Blue channel
5000×3333 × 8 bits

Raw size = pixels × channels × bits/channel = 16,665,000 × 3 × 8 = 399,960,000 bits
= 49,995,000 bytes ≈ 50.0 MB

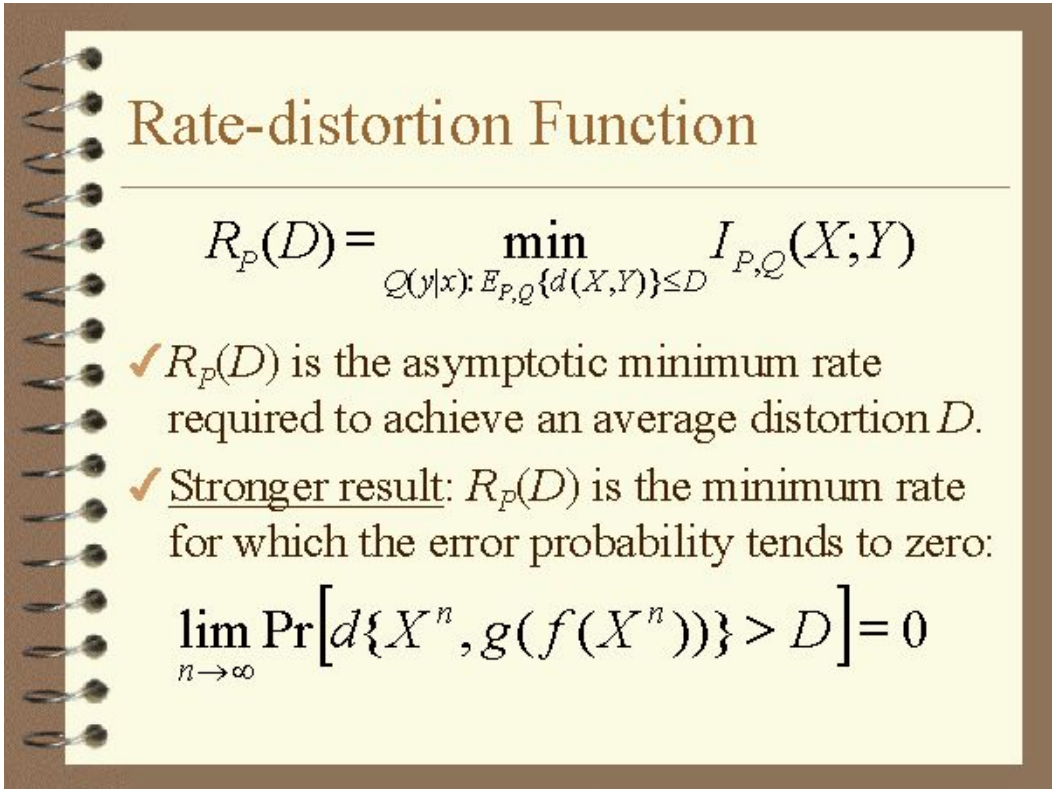
Each pixel = 3 numbers (R,G,B), each 0–255 (one byte). Stack all pixels → raw uncompressed size.

Rate-Distortion Function



Given a distortion level D, the minimum mutual information between source information X (distributed over a probability P) and reconstructed information Y (over a conditional probability distribution Q(y|x)) such as that the expected distortion given a function d over Q, P is less than D.

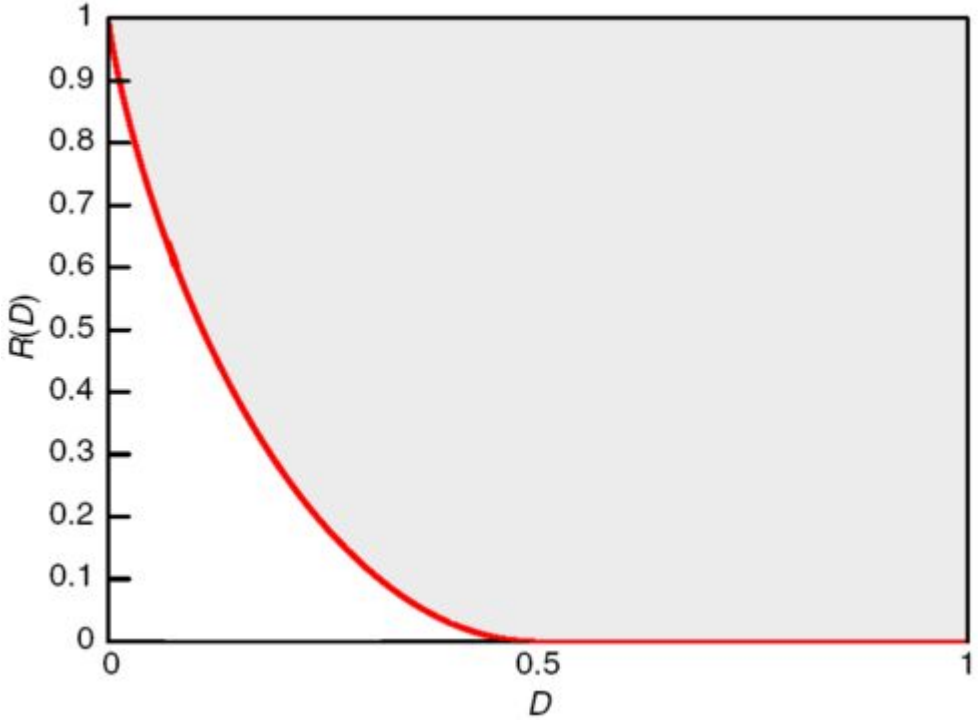
Rate-Distortion Function



In practice

- P, Q are unknown
- Distortion function d can be very complex
- Even if those are known, solving the optimization problem is very difficult.

Rate-Distortion Curve



- D: Distortion between source and target given a **distortion function**
- R(D): Number of bits needed to code the source information so that the distortion between target and source (under that specific distortion metric) is less than D.

Empirical Rate-Distortion Curve

- Compression Algorithm: JPEG
  - Different quality settings – trading off between image size and quality
- Metrics for Rate: file size, bits per pixel (bpp)
- A Metric for Distortion: Peak Signal-to-Noise Ratio (PSNR)
- To plot the RD curve:
  - Compress the original image under different quality settings
  - For each setting, plot its rate (size or bpp) and distortion (PSNR)

Empirical Rate-Distortion Curve

Original Image (50 MB)

Empirical Rate-Distortion Curve

JPEG Q90 (6.45 MB)

Empirical Rate-Distortion Curve

JPEG Q60 (2.65 MB)

Empirical Rate-Distortion Curve

JPEG Q35 (1.88 MB)

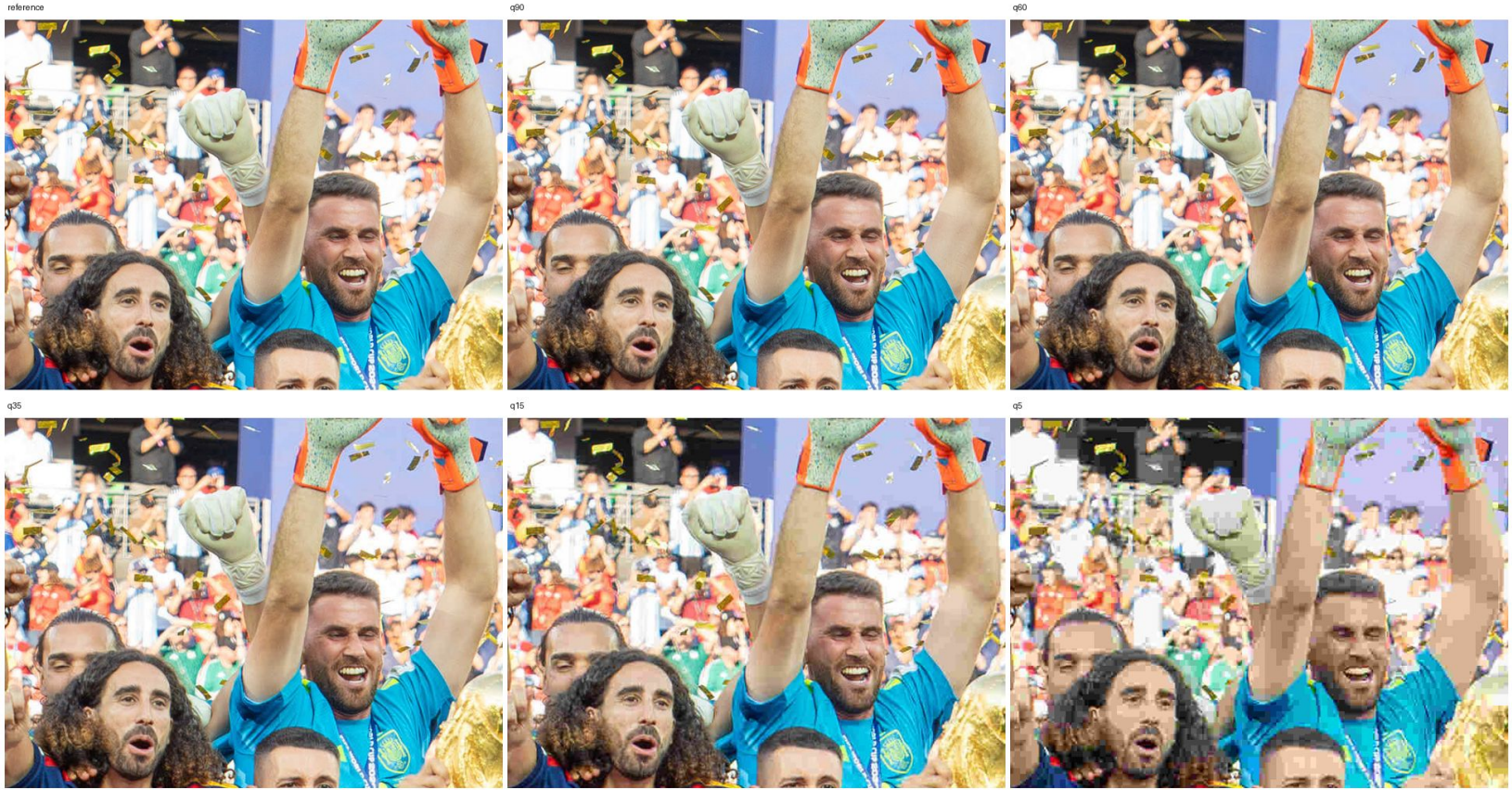
Empirical Rate-Distortion Curve

JPEG Q15 (1.16 MB)

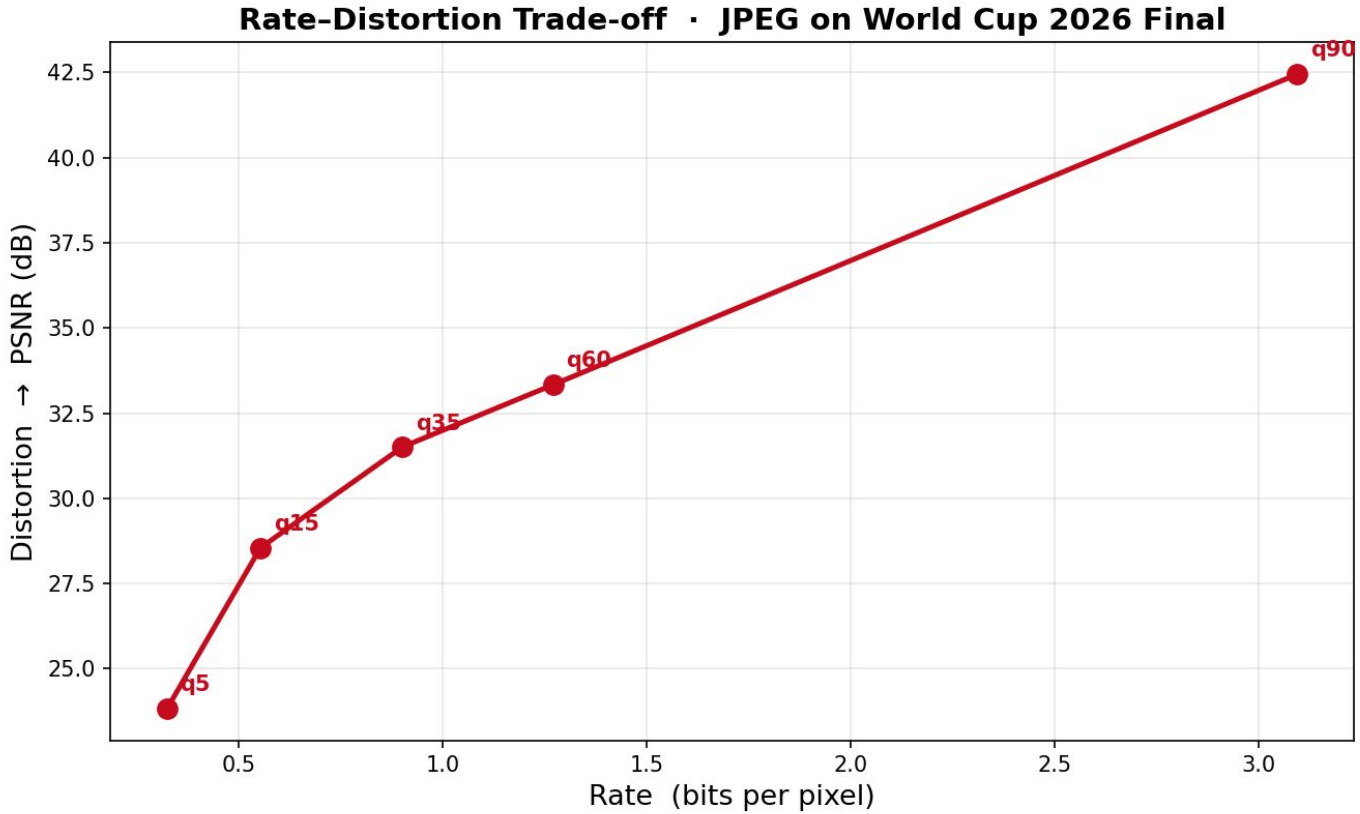
Empirical Rate-Distortion Curve

JPEG Q5 (0.68 MB)

Empirical Rate-Distortion Curve



Empirical Rate-Distortion Curve

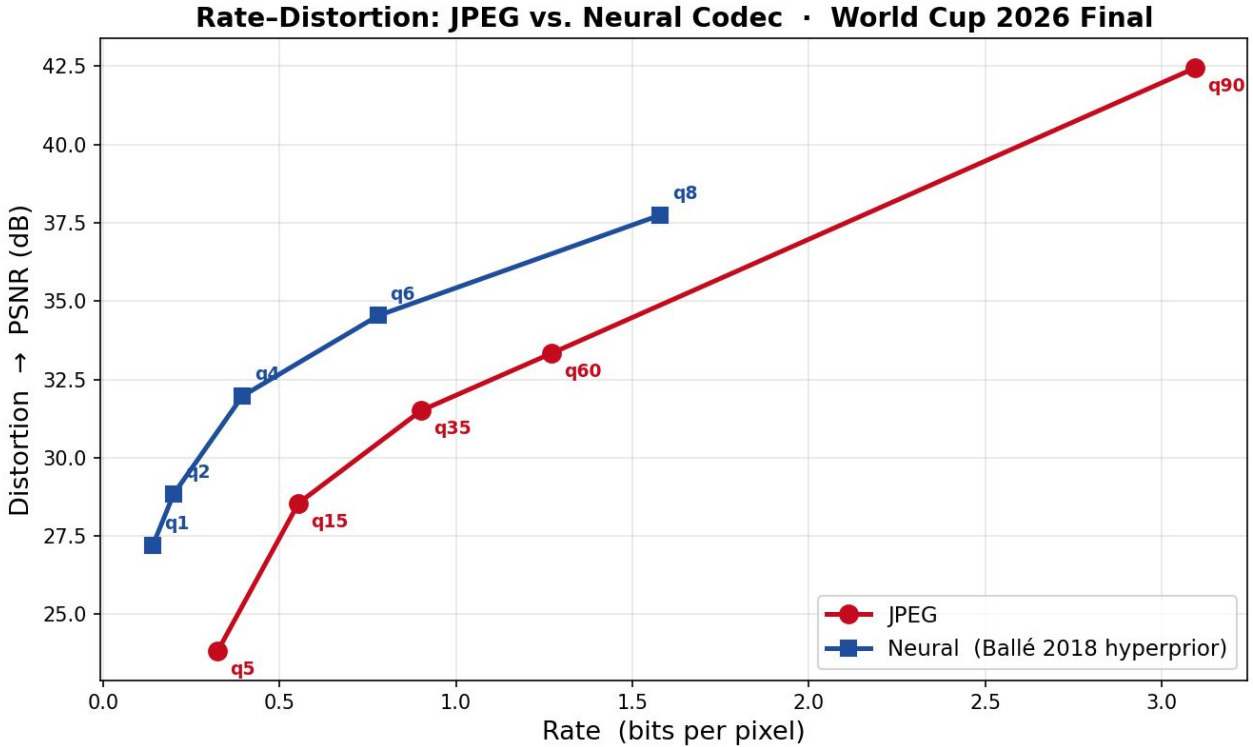


Empirical Rate-Distortion Curve

Machine Learning Based Image Compression

- Goal of lossy compression: using as few bits as possible to represent the source image as faithfully as possible
  - It's an optimization problem → data driven ML can help!
- There is a trade off rate and distortion – how do we formulate this problem as an objective for machine learning training?
- Answer: **Minimize R + λD**, where λ is a control parameter to characterize this trade-off
  - Higher λ → optimizing for **lower distortion over lower rate**
  - Lower λ → optimizing for **lower rate over lower distortion**

Empirical Rate-Distortion Curve



Empirical Rate-Distortion Curve

Reference (lossless)

JPEG · 0.55 bpp · 28.6 dB

43x smaller

Neural · 0.40 bpp · 32.0 dB

fewer bits, +3.4 dB

Summary

- Lossless Compression pipeline: Source data → encoder → bitstreams → decoder → reconstructed data
- How much we can compress a source depends on its probability distribution
  - Entropy is the quantity to characterize compressibility
- Lossy compression has an additional quantization step to reduce the number of bits needed to encode sources, and distortion is incurred
- Empirical rate-distortion curve is key to evaluate image compression algorithms.

What We Skipped & Future Exploration for You

- Prefix Free code: Ensuring no ambiguity when decoding a bitstream
- We assumed each unit of the source is independently and identically distributed (i.i.d.). What if they are not?
- More information theory terms: relative entropy, mutual information, etc
- Exact design of the image compression algorithm for JPEG and neural codec, etc.
- Image distortion metrics other than PSNR (MS-SSIM and LPIPS)
- Compression in 3D graphics (My current research!)
- Check out the following courses if any of these sounds interesting!
  - EE274: Data Compression
  - EE276: Information Theory
  - EE376C: Universal Information Processing

Advanced Topics: Kate Baker

Hardware Specialization and CUDA

Presentation by Kate Baker, Lvmin Zhang, and Connor Ding

Parallel Computing

In Stanford's CS149 (Parallel Computing) course, you learn about…

- Cache (coherence)
- Latency/bandwidth
- SPMD, SIMD, ISPC
- Barrier synchronization
- Work scheduling + distribution
- Shared address space
- CUDA
- RDDs
- Efficient convolution layers
- Hardware specialization
- Metapiplining
- Relaxed + transactional memory
- And more…

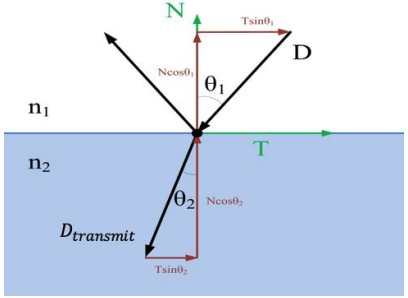
Parallel Computing

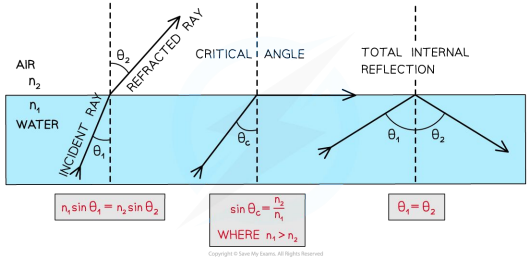
In Stanford's CS149 (Parallel Computing) course, you learn about…

- Cache (coherence)
- Latency/bandwidth
- SPMD, SIMD, ISPC
- Barrier synchronization
- Work scheduling + distribution
- Shared address space
- CUDA
- RDDs
- Efficient convolution layers
- Hardware specialization
- Metapiplining
- Relaxed + transactional memory
- And more…

Today, I'll introduce two topics: CUDA and hardware specialization.

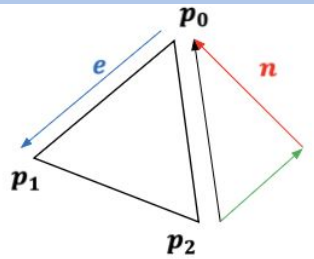
Why do we care about parallel computing?

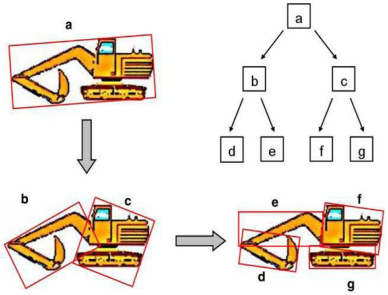


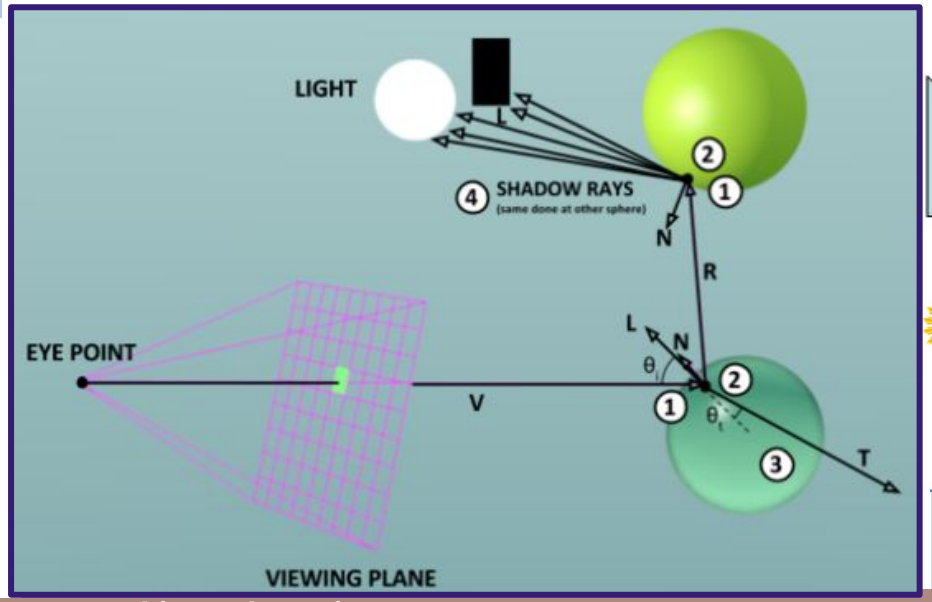


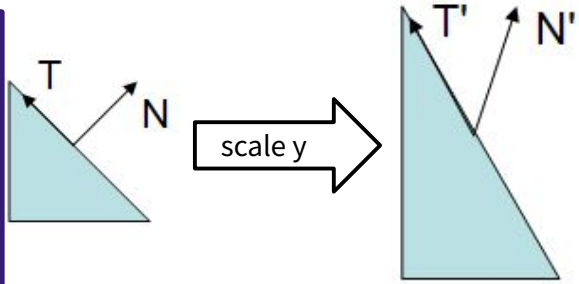
$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

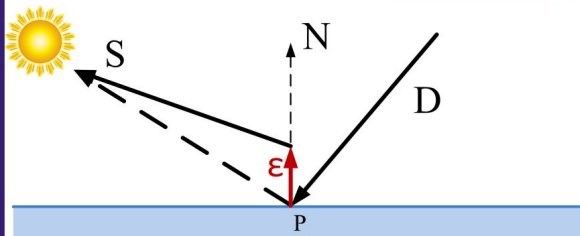
$$(u, v, A - P) \begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_o$$



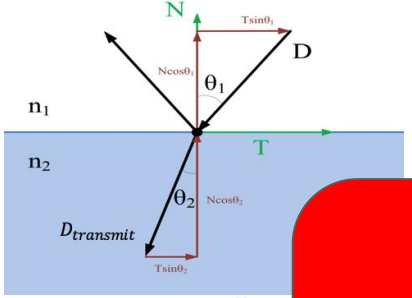


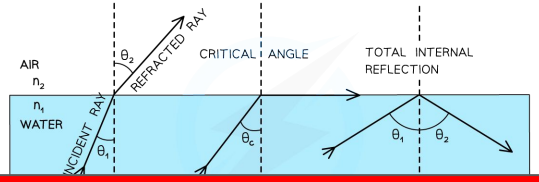






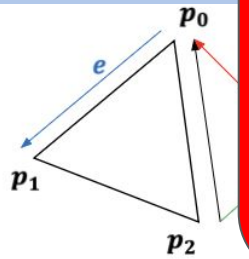
Why do we care about parallel computing?





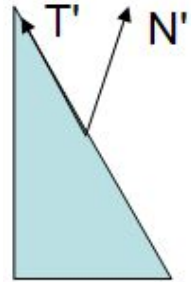
$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

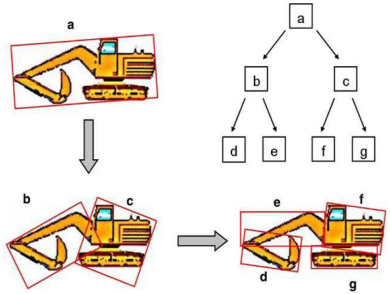
$$(u, v, A - P)\begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_o$$

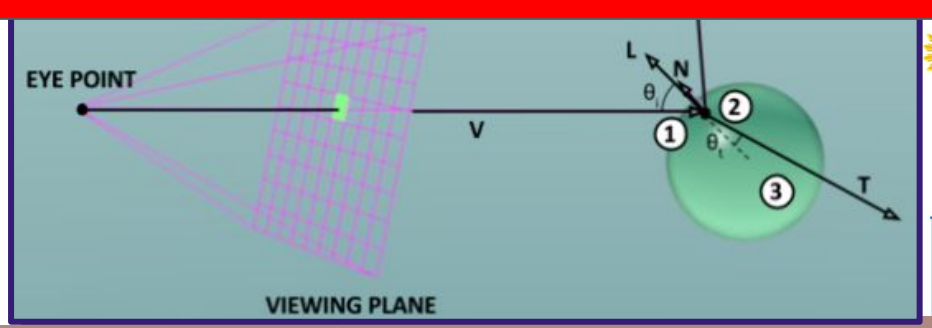


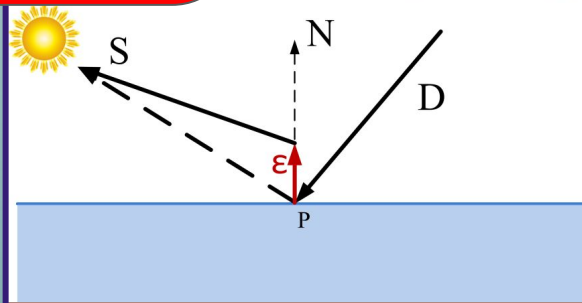
Ray tracing takes SO MUCH WORK!

cale y



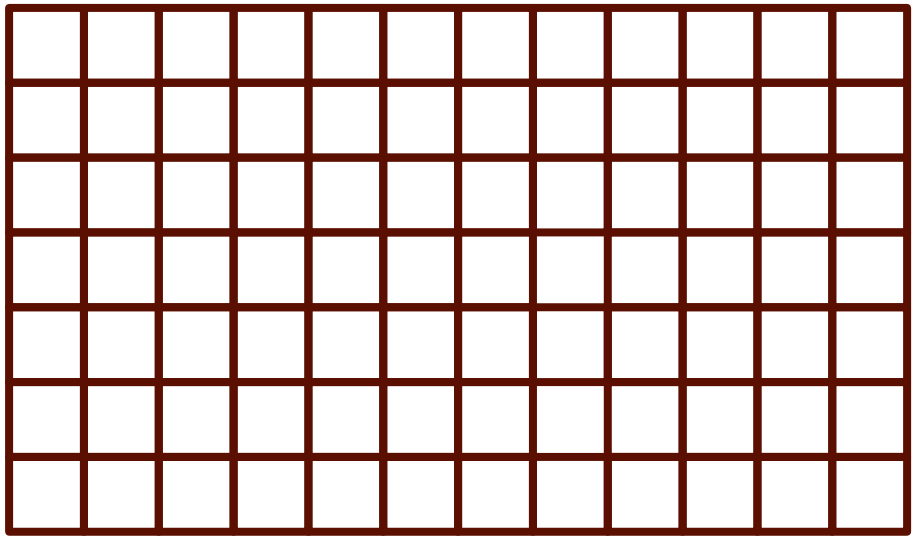






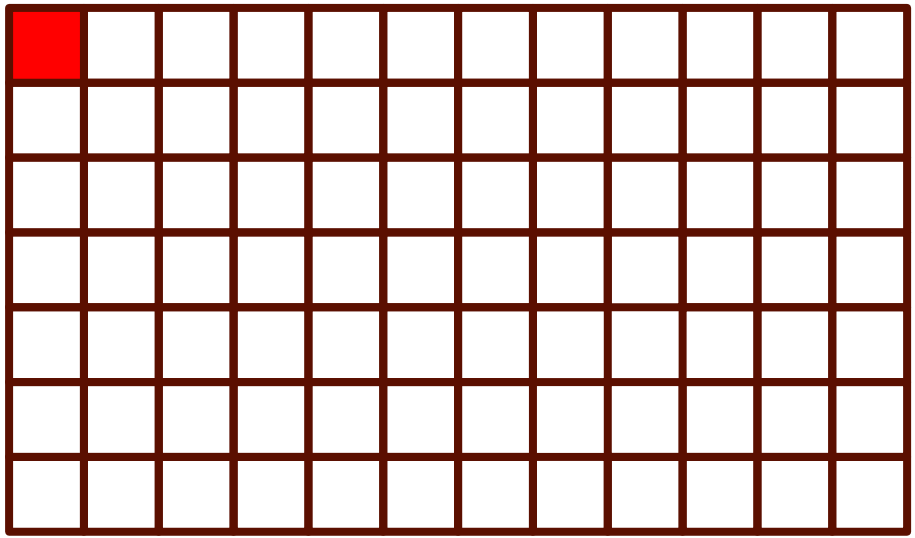
Why do we care about parallel computing?

Sequential ray tracing…



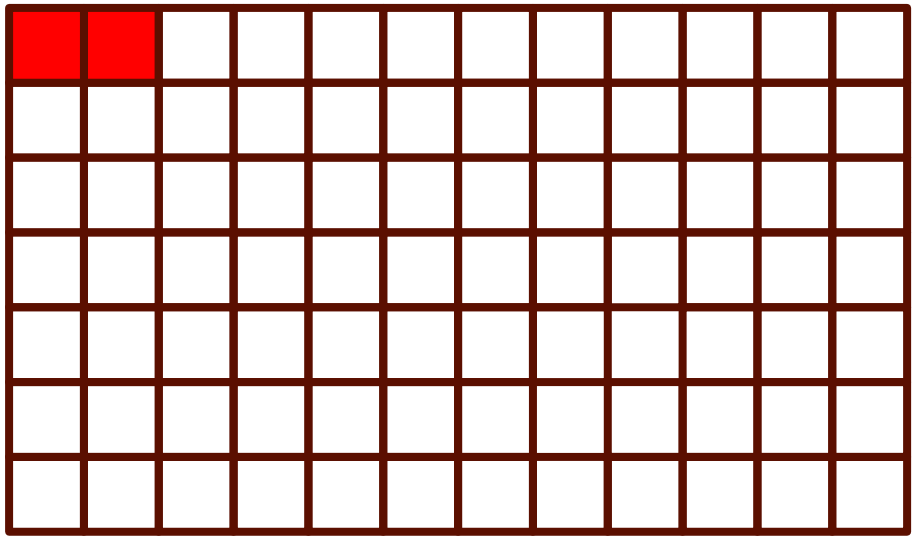
Why do we care about parallel computing?

Sequential ray tracing…



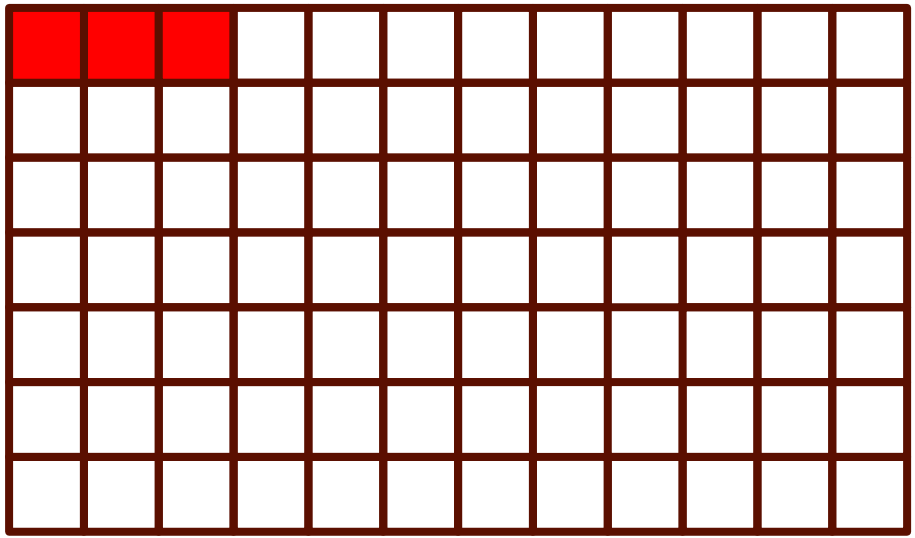
Why do we care about parallel computing?

Sequential ray tracing…



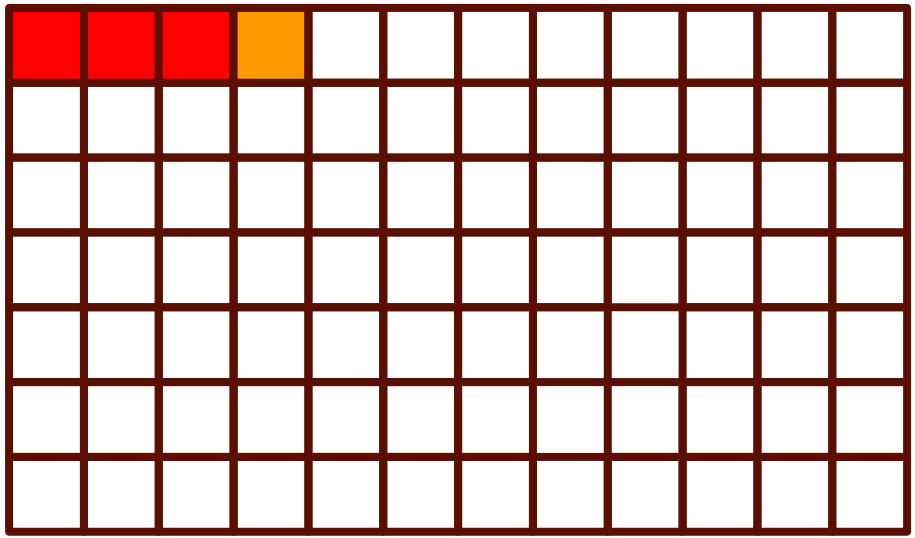
Why do we care about parallel computing?

Sequential ray tracing…



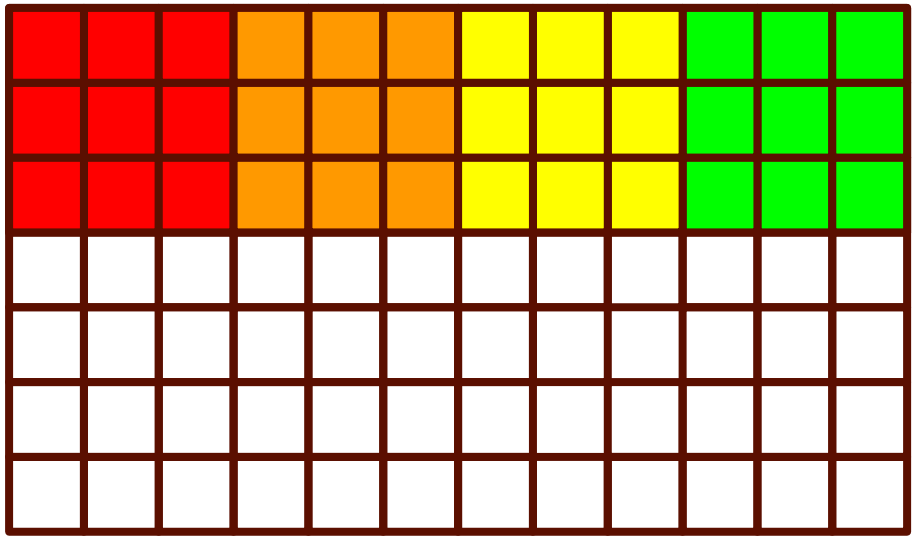
Why do we care about parallel computing?

Sequential ray tracing…



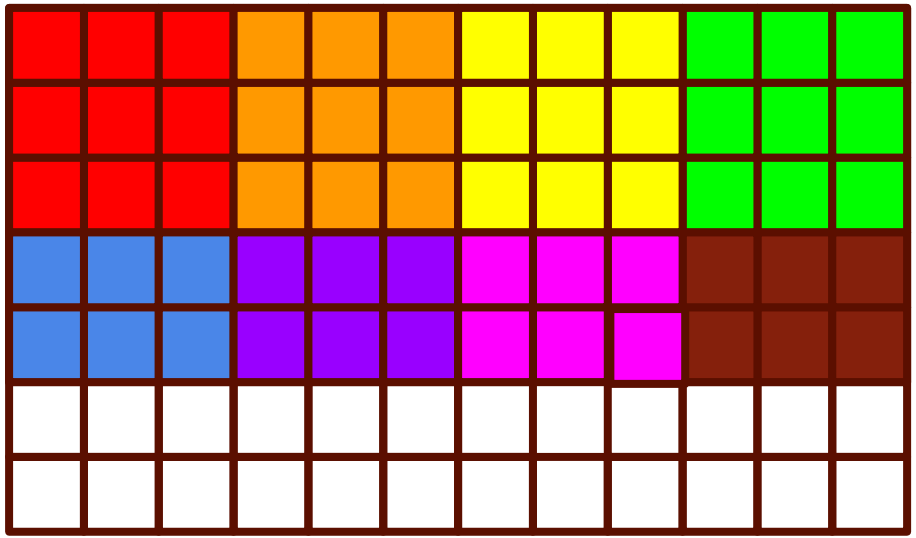
Why do we care about parallel computing?

Sequential ray tracing…



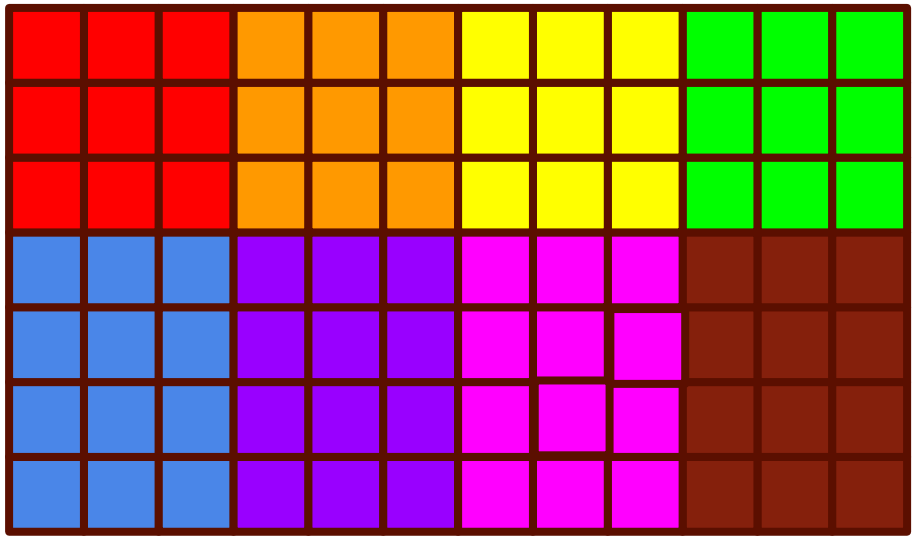
Why do we care about parallel computing?

Sequential ray tracing…



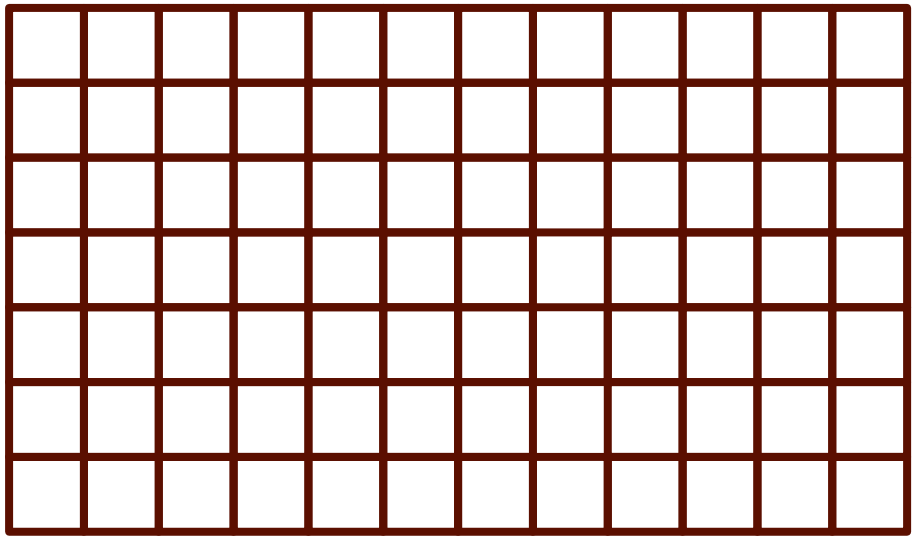
Why do we care about parallel computing?

Sequential ray tracing… after a while…



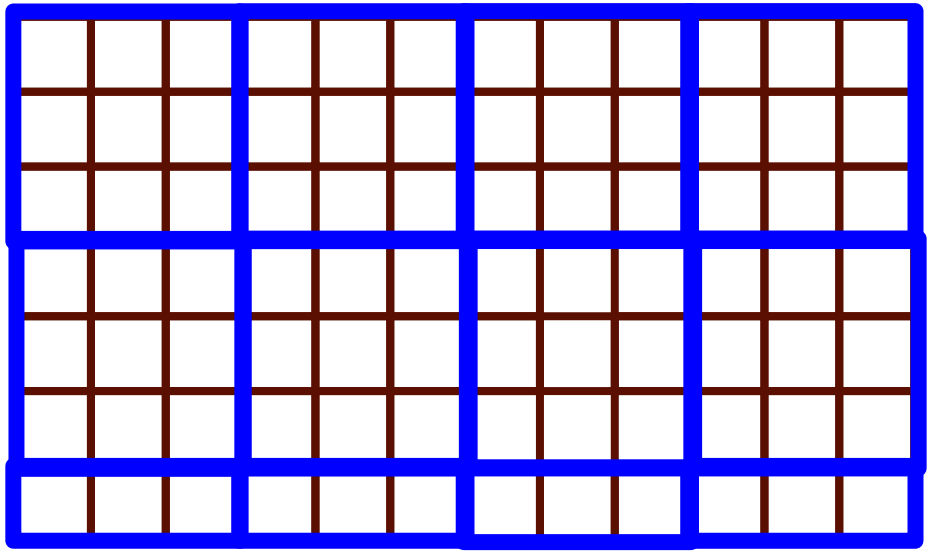
Why do we care about parallel computing?

Versus parallel ray tracing…



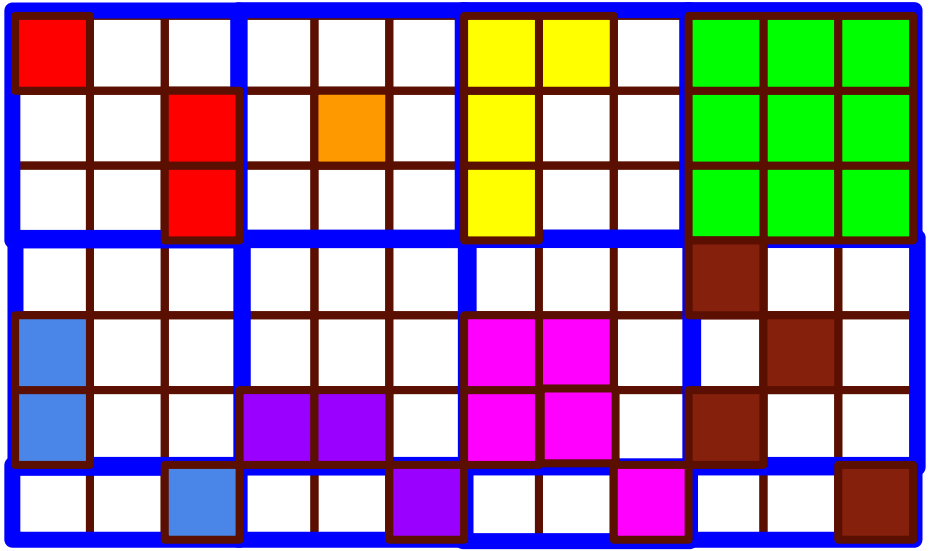
Why do we care about parallel computing?

Versus parallel ray tracing…



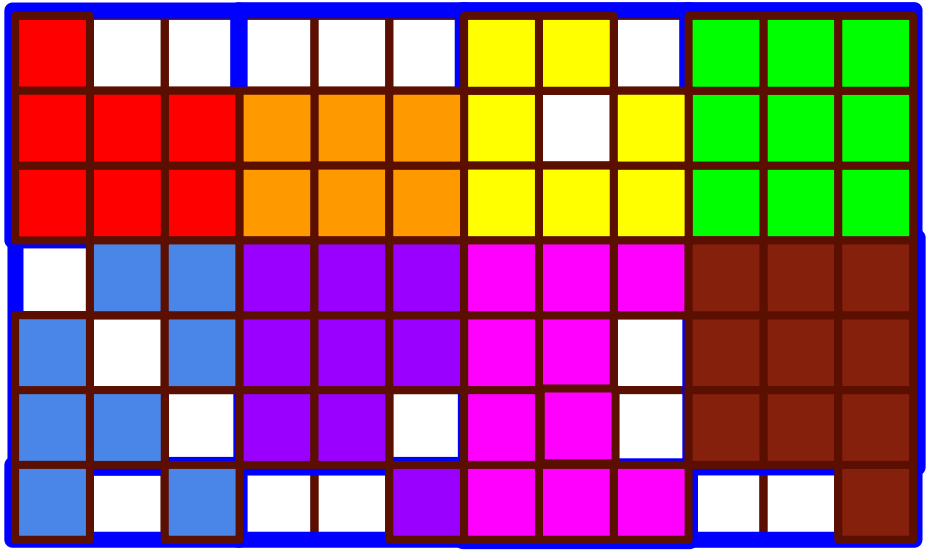
Why do we care about parallel computing?

Versus parallel ray tracing…



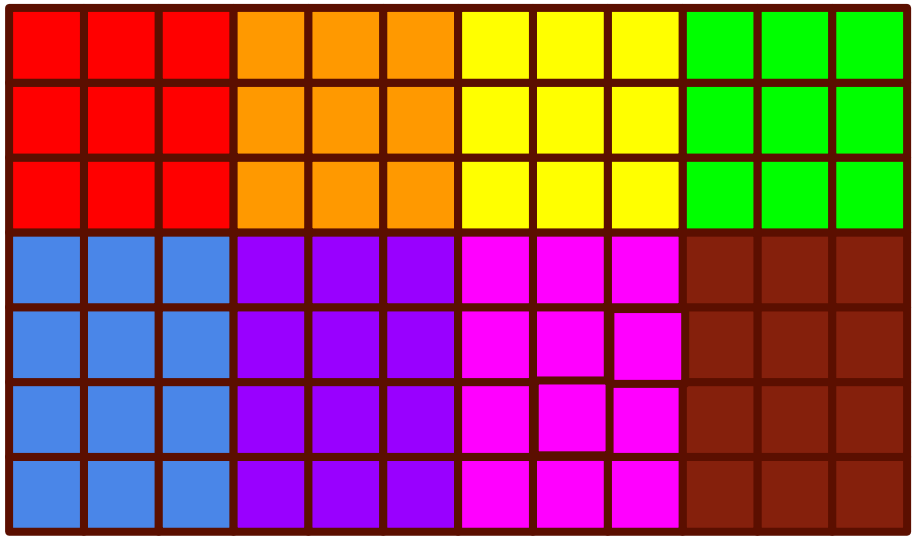
Why do we care about parallel computing?

Versus parallel ray tracing…



Why do we care about parallel computing?

Versus parallel ray tracing… all done!



Why do we care about parallel computing?

- If we can do ray tracing fast, then we can support **real time rendering**
  - Useful for video games, scientific visualization, etc.

- Also allows us to provide real time rendering for many **different computers** and machine capabilities
  - E.g. cloud, native deployments, smaller/cheaper hardware

Parallel Computing: Basics

- A **parallel computer** is a collection of processing elements that cooperate to solve problems quickly

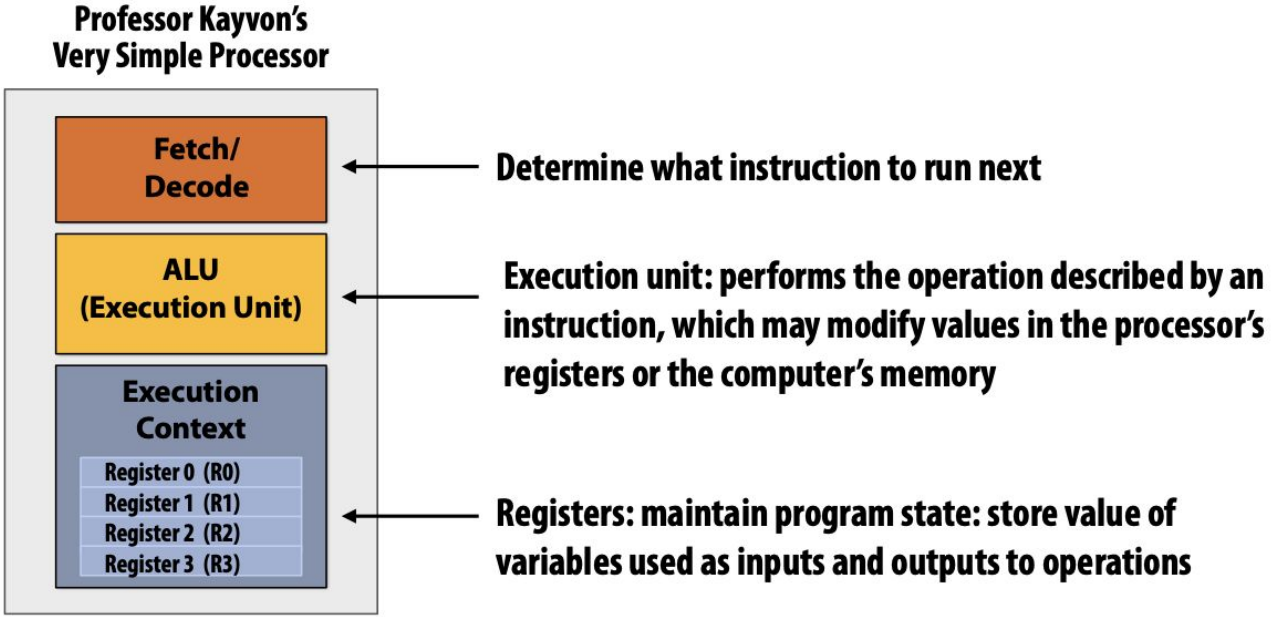
Parallel Computing: Basics

- A **parallel computer** is a collection of processing elements that cooperate to solve problems quickly

- **Programmer perspective**: make use of provided machine capabilities in most efficiency way
  - E.g. time/space complexity (Big O)
- **Hardware designer perspective**: choosing right capabilities to put in a system

Hardware: Basics

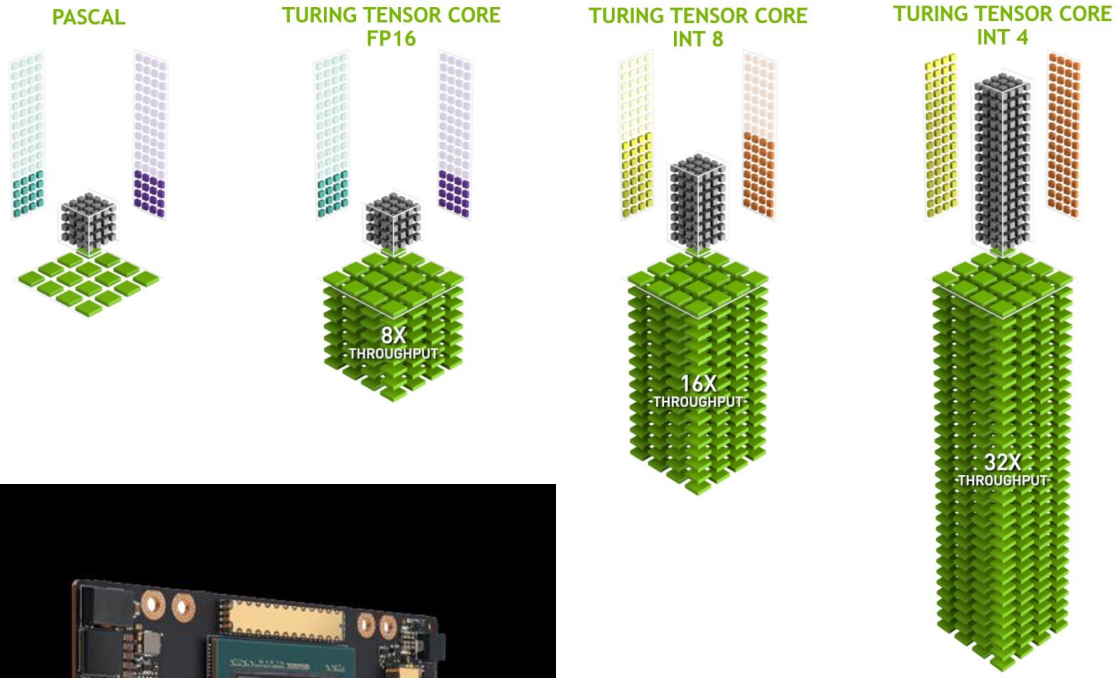
A single **core** executes one instruction on one piece of data per cycle

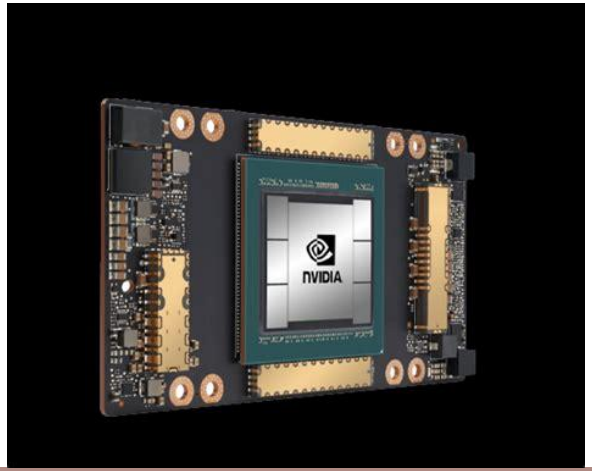


Hardware: Basics

Tensor Cores

- AI specialized cores
- Great at matrix multiplications
- ^ Great for AI (especially CNN) work





Hardware: Threads

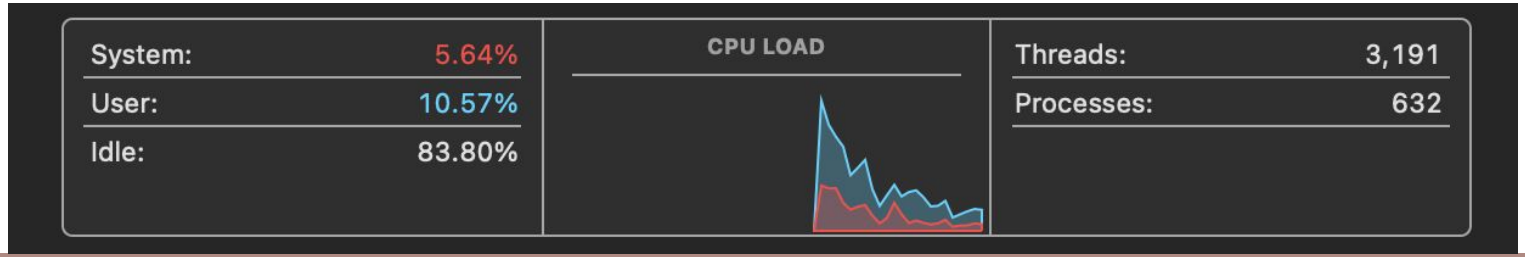
Thread

- A single sequence stream in a process
- **Process**
  - Independent running program with many threads
  - Private memory space
- **GPU thread (a warp)**
  - Runs many threads concurrently to perform small piece of work

Hardware: Threads

Thread

- A single sequence stream in a process
- **Process**
  - Independent running program with many threads
  - Private memory space
- **GPU thread (a warp)**
  - Runs many threads concurrently to perform small piece of work
- Activity Monitor: Can check threads/processes on computer:



Hardware: Threads

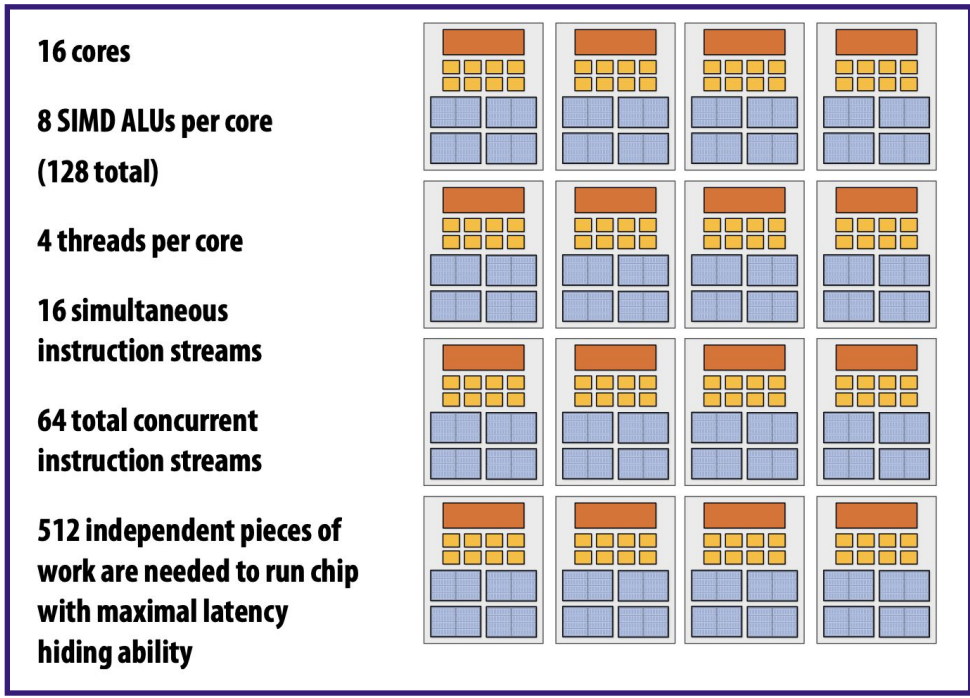
Threads + Shared Memory

- Threads typically operate in **private address space**
  - Needs a message passing model and way to **schedule** them
- **Deadlock** occurs when a thread is waiting to receive but there is no send

Hardware: Chips

Fictitious multicore chip

- One **core** → executes one instruction on one piece of data per cycle
- **SIMD** → core can operate on 8 numbers at once with a single instruction
- **Multicore** → 16 cores x 8 ALU lanes = 128 ALUs

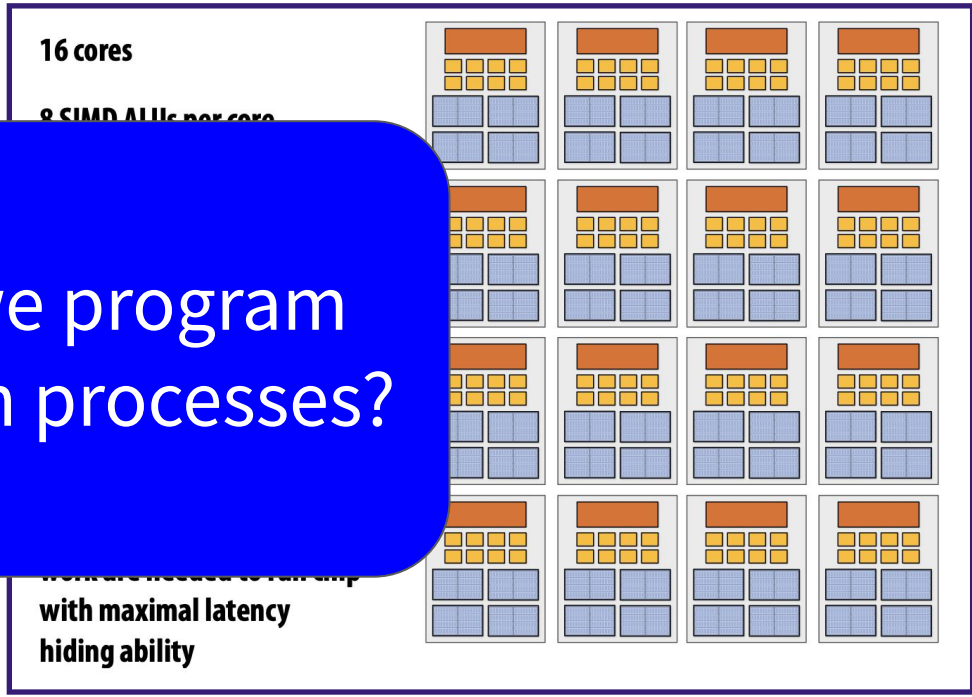


Hardware: Chips

Fictitious multicore chip

- One **core** → executes one instruction on or
  per cycle
- **SIMD** → core car
  numbers at onc
  instruction
- **Multicore** → 16
  lanes = 128 ALUs

How do we program these batch processes?



Hardware: Chips

Fictitious multicore chip

- One **core** → executes one instruction on or per cycle
- **SIMD** → core car numbers at onc instruction
- **Multicore** → 16 lanes = 128 ALUs

With CUDA!

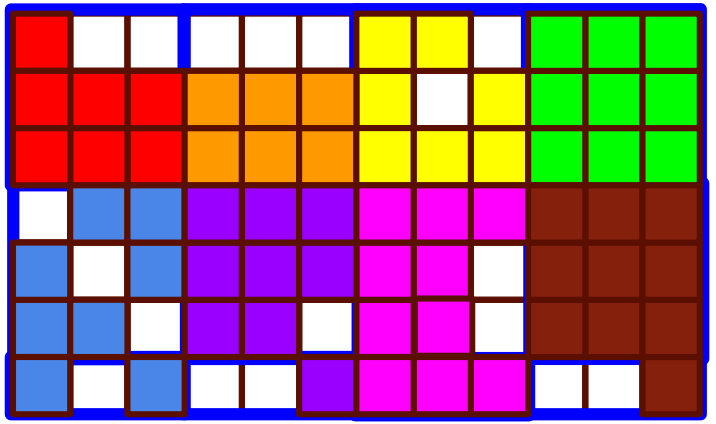
CUDA: Basics

- CUDA = Compute Unified Device Architecture
- Parallel computing platform + programming model (an API) developed by NVIDIA for use with NVIDIA GPUs
- Programmers can write code in languages like C, C++, and Python

CUDA: Basics

- CUDA = Compute Unified Device Architecture
- Parallel computing platform + programming model (an API) developed by NVIDIA for use with NVIDIA GPUs
- Programmers can write code in languages like C, C++, and Python

- With CUDA, you can harness the power of GPUs!
  - Can program many processes to run the same instruction simultaneously
  - This **dramatically accelerates rendering techniques** like ray tracing



CUDA: Kernels

- Used to program the GPU: **GPU = device & CPU = host**
  - Aka CPU launches kernels on the GPU

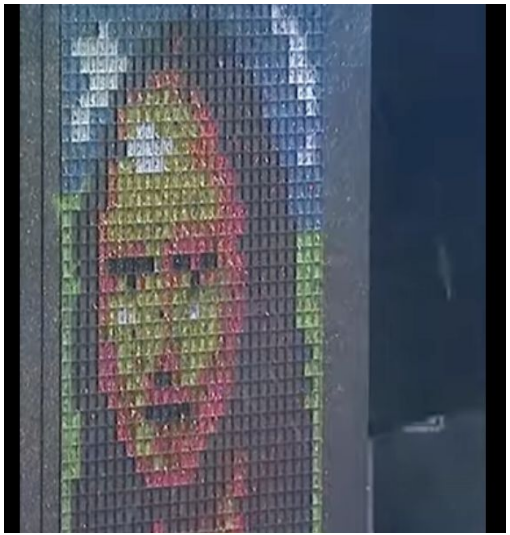
CUDA: Kernels

- Used to program the GPU: **GPU = device & CPU = host**
  - Aka CPU launches kernels on the GPU

- **Kernels** run once per thread, across potential thousands of threads simultaneously

CUDA: Kernels

- **Kernels** run once per thread, across potential thousands of threads simultaneously

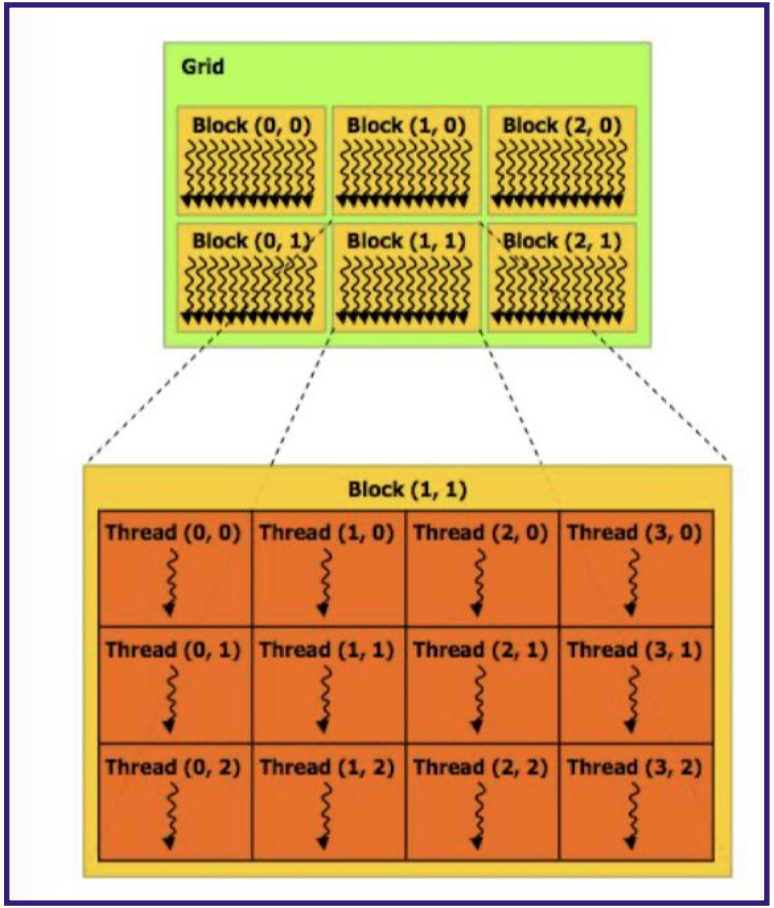


Recall CPU vs GPU Paintball Showdown from week 1!

https://www.youtube.com/watch?v=DZEYmAp-Oy0

CUDA: Design

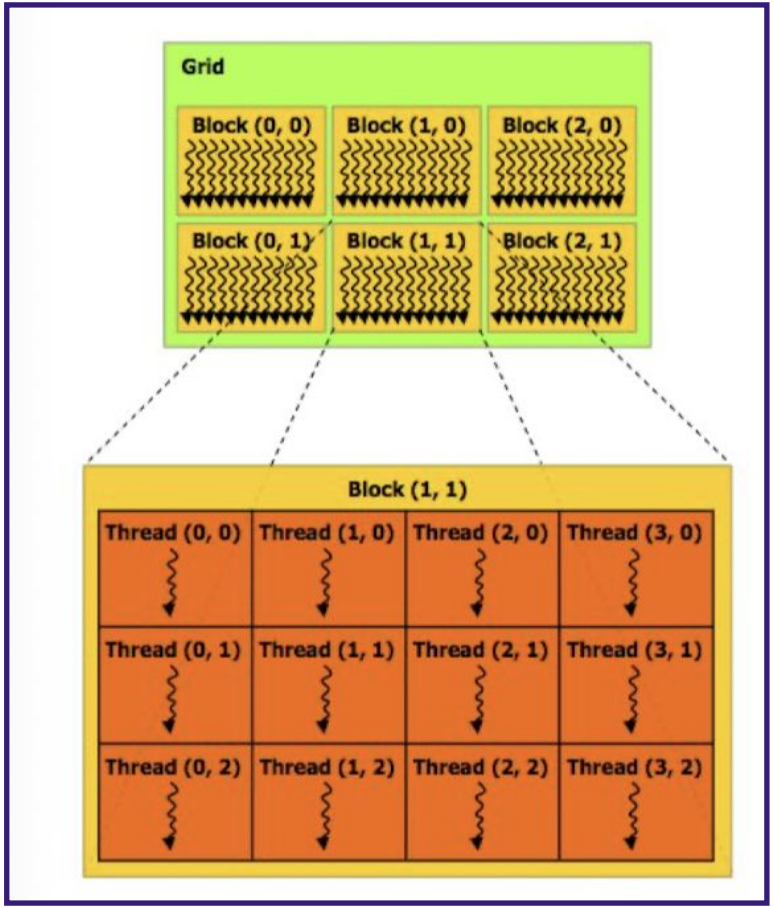
- **Grids** are made up of **blocks** and blocks are made up of **threads**



CUDA: Design

- **Grids** are made up of **blocks** and blocks are made up of **threads**

- Every thread figures out who it is using built in variables
  - `blockIdx` → which block
  - `blockDim` → how big is the block
  - `threadIdx` → which thread is it within the block (can be 3D)



CUDA: Design

- `myKernel<<<NumBlocks,threadsPerBlock>>>(args);`
  - Aka run this function across this many blocks, each with this many threads
  - `<<<...>>>` is CUDA's special launch syntax

CUDA: Example

```
const int nColsImage = 12;
const int nRowsImage = 6;
```

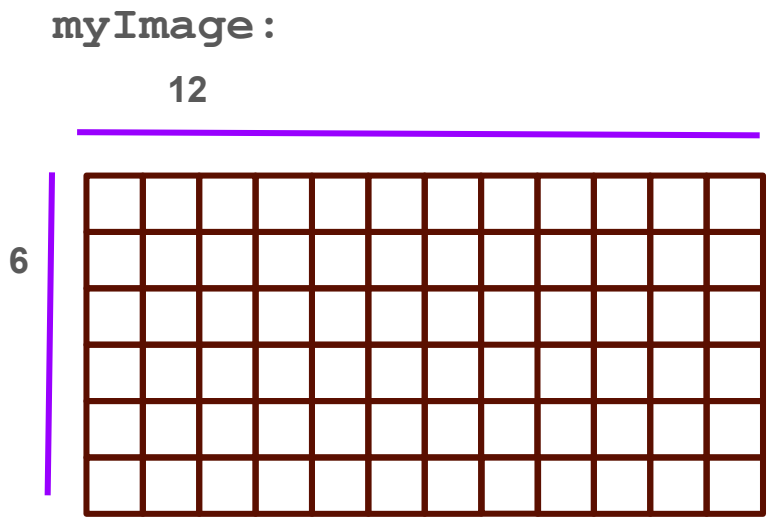
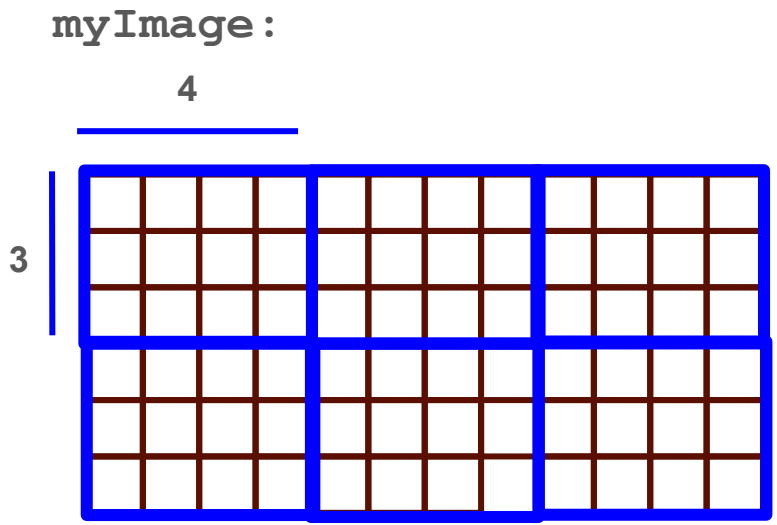


Figure out how much work you might do (e.g. # of pixels for ray tracing an image)

CUDA: Example

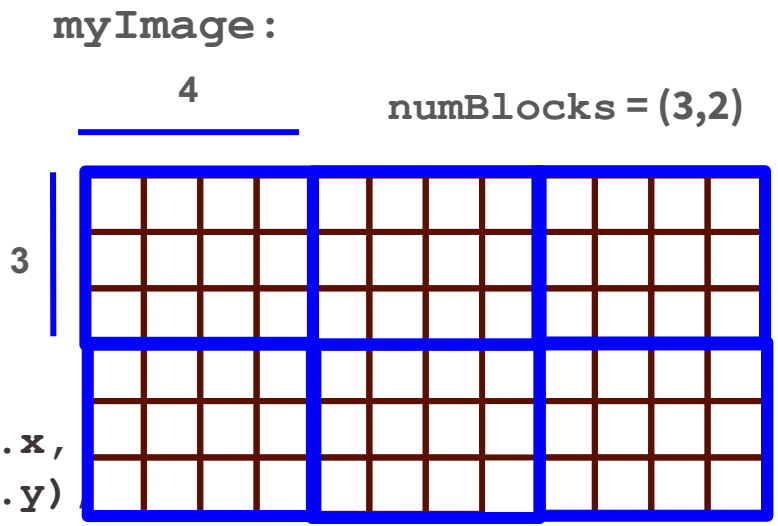
```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
```



Determine how you want to define the blocks of work. Here, we have blocks of **dimension 4x3**

CUDA: Example

```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);
```

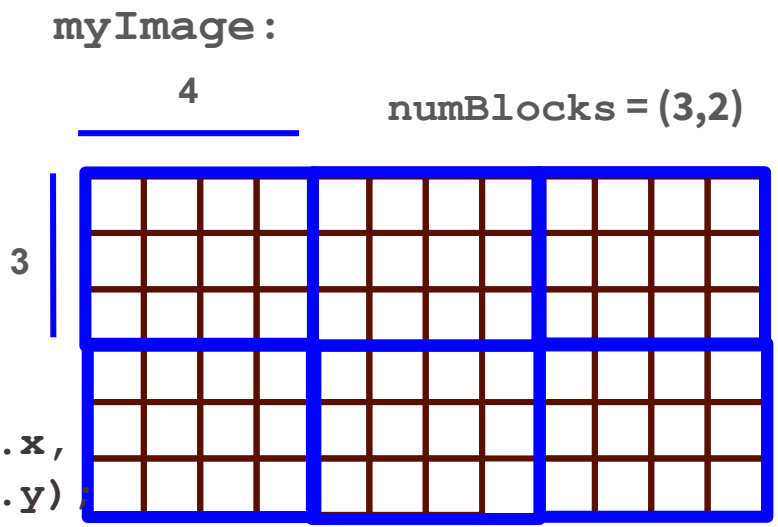


Use this information to figure out how many blocks of your defined size are needed. In this case, **`numBlocks` = (3,2)** which means there are **6 blocks**.

CUDA: Example

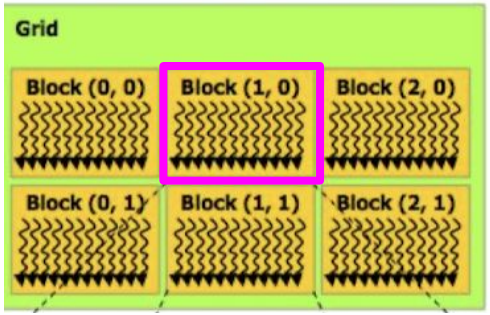
```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);
```

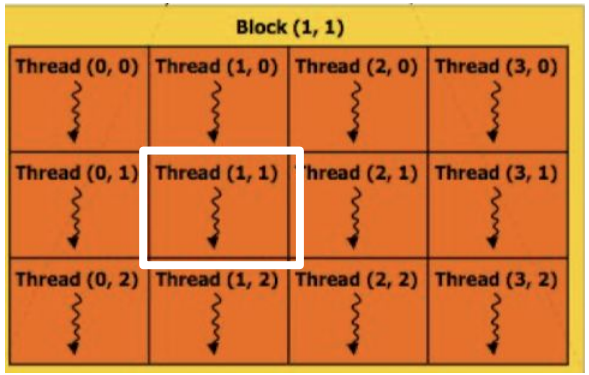
```
__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // ... ray tracing stuff using x, y ...
}
```

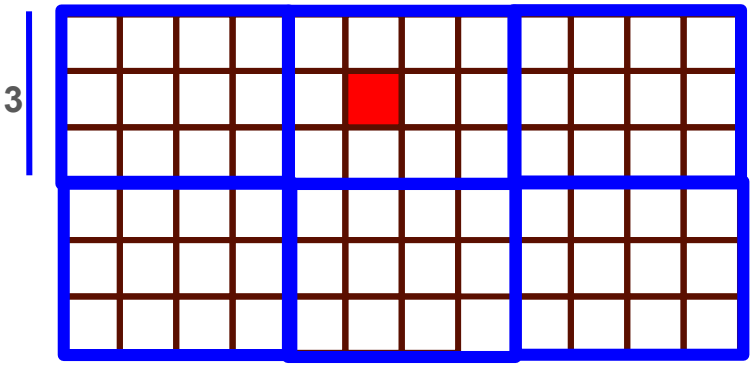


Now, ensure you have a **kernel** (CUDA-compatible function) that you're prepared to call many, many times. Use the built-in indices to determine the Pixel (x,y) that should be used

CUDA: Example







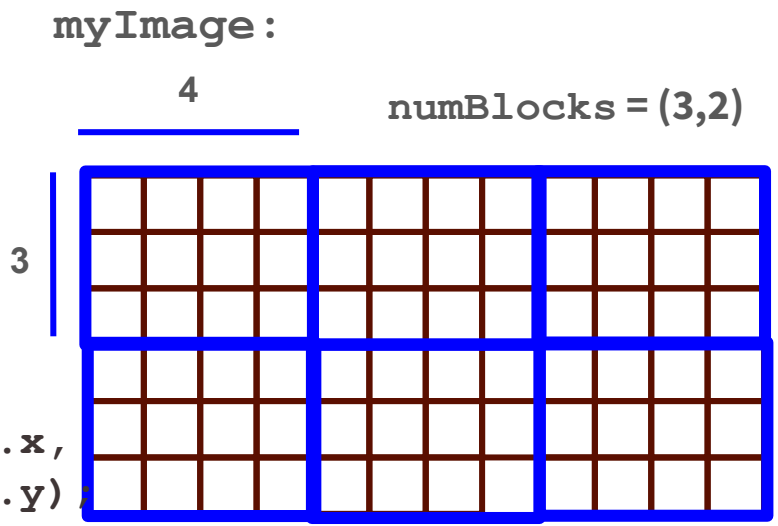

```
__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // ... ray tracing stuff using x, y ...
}
```

Example: red pixel above.

x = 1 * 4 + 1 = 5
y = 0 * 3 + 1 = 1

So, we've computed **pixel (5,1)**

CUDA: Example

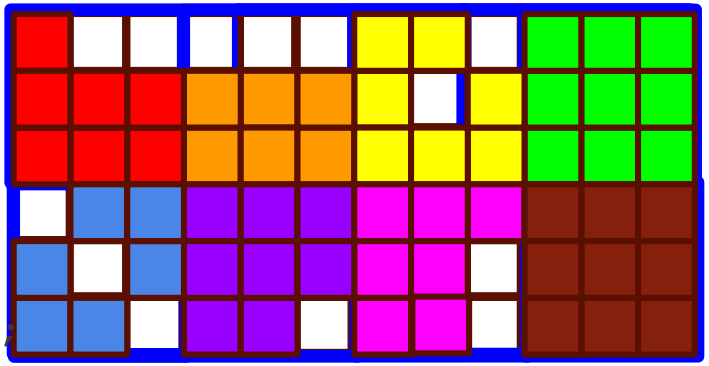



```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);

__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // ... ray tracing stuff using x, y ...
}

rayTrace<<<numBlocks, threadsPerBlock>>>(myImage);
```

Then launch your kernel! 🚀

CUDA: Example

myImage:

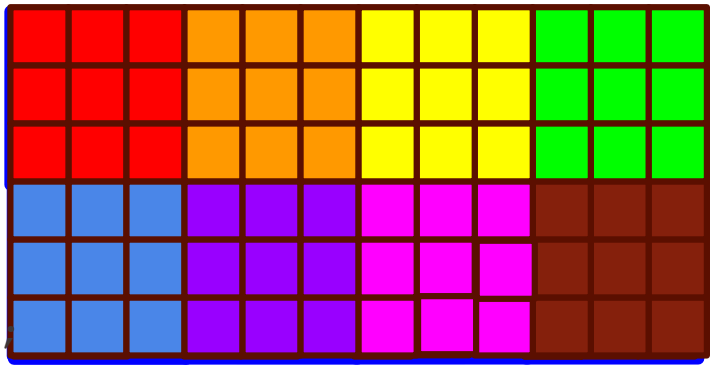


```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);

__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // ... ray tracing stuff using x, y ...
}

rayTrace<<<numBlocks, threadsPerBlock>>>(myImage);
```

Now each pixel operation can run **independently...**

CUDA: Example

myImage:

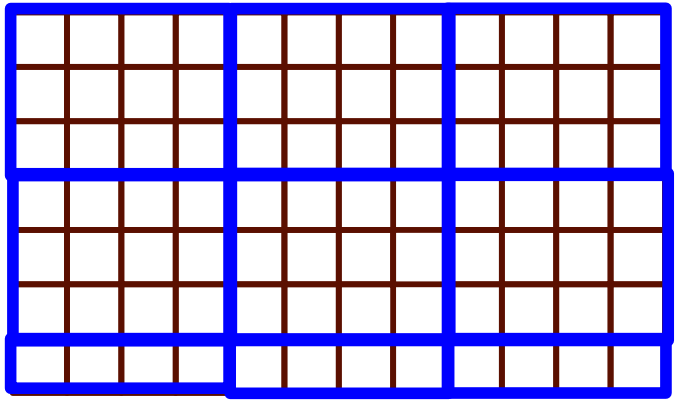


```
const int nColsImage = 12;
const int nRowsImage = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);

__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // ... ray tracing stuff using x, y ...
}

rayTrace<<<numBlocks, threadsPerBlock>>>(myImage);
```

Now each pixel operation can run **independently… until it's done!**

CUDA: Example

myImage:



```
const int nColsImage = 12;
const int nRowsImage = 7;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(nColsImage/threadsPerBlock.x,
               nRowsImage/threadsPerBlock.y);

__global__ void rayTrace(Image myImage){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= nColsImage || y >= nRowsImage) return;
    // ... ray tracing stuff using x, y ...
}

rayTrace<<<numBlocks, threadsPerBlock>>>(myImage);
```

Ensure you have a bounds check in case you have any uneven blocks!

CUDA: Error Catching

- Every CUDA API call returns a `cudaError_t` to catch (potential) issues
  - `cudaGetLastError()`
  - `cudaGetErrorString()`
- Common issues: out of bounds, wrong launch number

CUDA: Ray Tracing Relevance

- Check this out
- One thread renders one pixel

CUDA: Ray Tracing Relevance

- Check this out
- One thread renders one pixel
- Kernel determines which pixel it owns using `blockIdx` / `threadIdx`

CUDA: Ray Tracing Relevance

- Check this out
- One thread renders one pixel
- Kernel determines which pixel it owns using `blockIdx` / `threadIdx`
- Grids and blocks are launched in 2D so that thread's (x,y) maps to a pixel's (x,y) position
  - We need to have bounds checks because image dimensions rarely divide evenly into block sizes

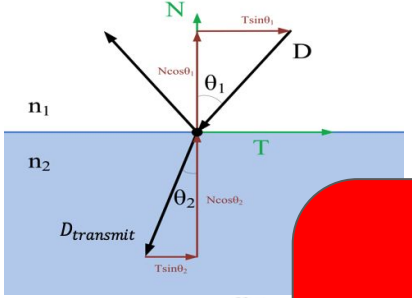
CUDA: Ray Tracing Relevance

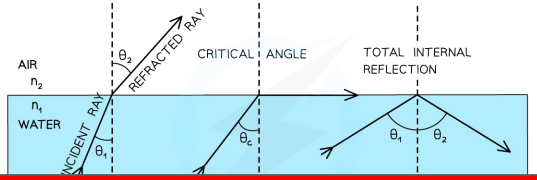
- Block size matters
  - In RT, choose small square blocks (e.g. 8x8 threads – a multiple of 32) so that neighboring pixels (which tend to have similar workloads) land in the same warp
  - There may be some sequentialization if one thread completes task well before the others in its warp

CUDA: Ray Tracing Relevance

- Block size matters
  - In RT, choose small square blocks (e.g. 8x8 threads – a multiple of 32) so that neighboring pixels (which tend to have similar workloads) land in the same warp
  - There may be some sequentialization if one thread completes task well before the others in its warp
- Modern NVIDIA GPUs have **RT cores** to accelerate 2 expensive RT tasks:
  - Traversing BVHs
  - Ray-triangle intersection tests
  - A CUDA thread async launches another program in the RT core for these tasks, then the original does another task

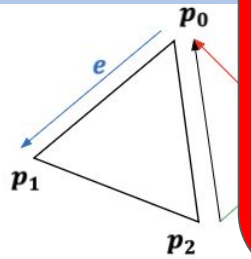
Yay parallel computing!





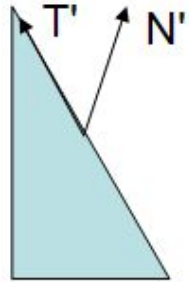
$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

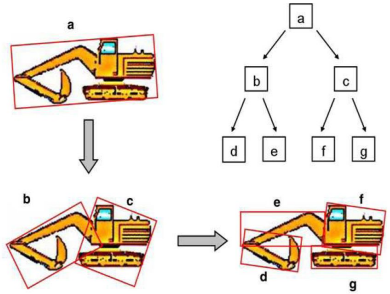
$$(u, v, A - P)\begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_o$$

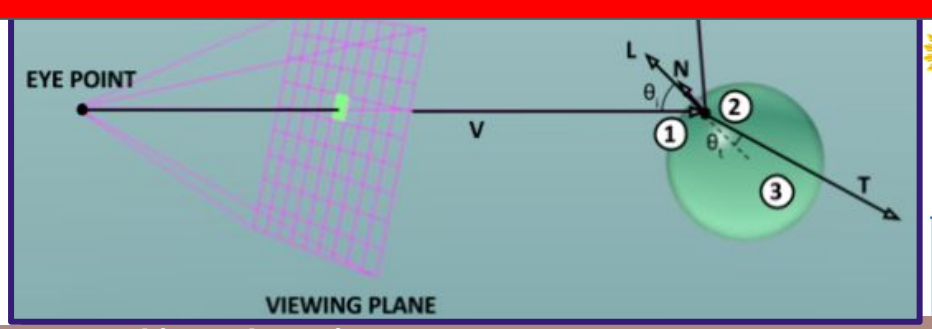


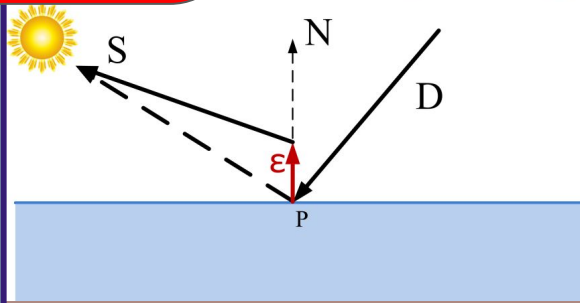
We can do this much faster!

cale y









Some FAQs

Additional material!

Why not launch one giant block covering the whole image?

- We have the 1024 thread limit per block and it would be less “parallel”

Does the block size matter?

- Yes! Best bet is to pick a size that’s a multiple of 32 to avoid wasting part of a warp and to pick smaller blocks to keep workloads similar

What is AI-based denoising?

- It helps clean up images rendered with too few rays per pixel to be noise free (typically done after the ray tracing

What about additional rays for each pixel?

- This tech can use advanced scheduling schemes to separate ray generation, intersection, shading, etc. into different kernels

Some FAQs

Additional material!

What are integrated GPUs vs. gaming servers?

- Integrated (native) GPUs are graphics processors built directly into the same chip as the CPU (e.g. Apple's GPU inside the M-series chip)
- Gaming/cloud servers use large/powerful/dedicated GPUs that sit in a data center. More complex rendering is done on these devices and then streamed to your local machine.

What about multiple rays per pixel for anti-aliasing?

- This is called supersampling
- Typically, there is still one thread per pixel, but that thread performs N samples sequentially and then averages them
- ^This aligns with the thread having private address/data spaces

Ray Tracing Optimization: Performance

Additional material!

- **RT Core count**
  - More cores = better performance
  - Each specialized unit can handle complex calculations
- **Tensor cores**
  - AI-driven denoising and upscaling
- **Clock speed and memory**
  - Higher clock speeds + more memory reduce bottlenecks
- **Software Optimization**
  - Leverage smart upscaling techniques

CUDA: Kernels

Additional material!

- Kernels **run async** with host side code
  - Aka the CPU keeps running code immediately after launching
  - If you need the GPU to finish before the CPU continues, then call `cudaDeviceSync()`

Hardware: Basics

Additional material!

Core Speed	3592.72 MHZ
Multiplier	X 36.0
Bus Speed	99.80 MHz

Clock speed

- A measure of how fast the CPU can execute instructions
- Generates timing pulses for operations
- A frequency (measured in hertz)
  - 1 MHz = 1 mil cycles/sec
  - 1 GHz = 1 bil cycles/sec

Core Speed = (Bus Speed)*(Mutliplier)

Core Speed = (99.80MHz)*(36.0) = 3592.8MHz = 3.50GHz

Hardware: Threads

Threads + Shared Memory

- Threads typically operate in **private address space**
  - Needs a message passing model and way to **schedule** them
- **Deadlock** occurs when a thread is waiting to receive but there is no send
- We need **storage systems** in case processes fail
  - E.g. **Hadoop DFS** where files are broken into blocks, and each block is replicated on multiple datanodes
- **Apache Spark**
  - Open source analytics engineer for processing large amounts of data + fault tolerance

Additional material!

Hardware: Basics

Additional material!

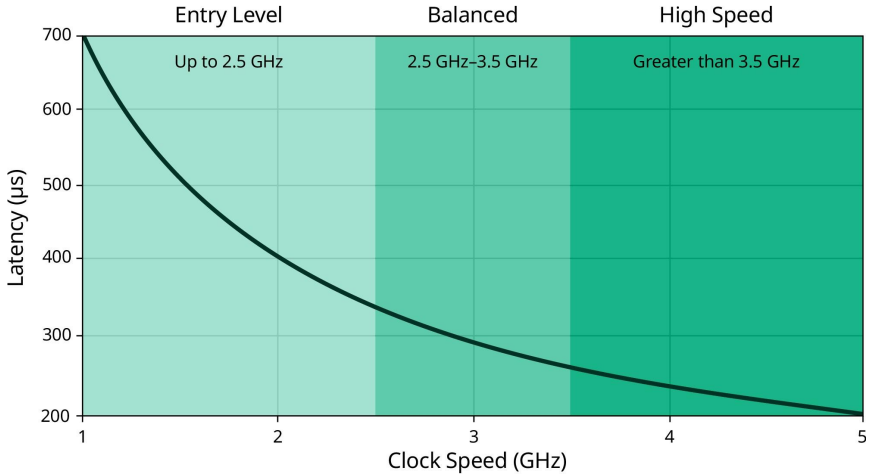
Core Speed	3592.72 MHZ
Multiplier	X 36.0
Bus Speed	99.80 MHz

Clock speed

- A measure of how fast the CPU can execute instructions
- Generates timing pulses for operations
- A frequency (measured in hertz)
  - 1 MHz = 1 mil cycles/sec
  - 1 GHz = 1 bil cycles/sec
- Diminishing returns

Core Speed = (Bus Speed)*(Mutliplier)

Core Speed = (99.80MHz)*(36.0) = 3592.8MHz = 3.50GHz

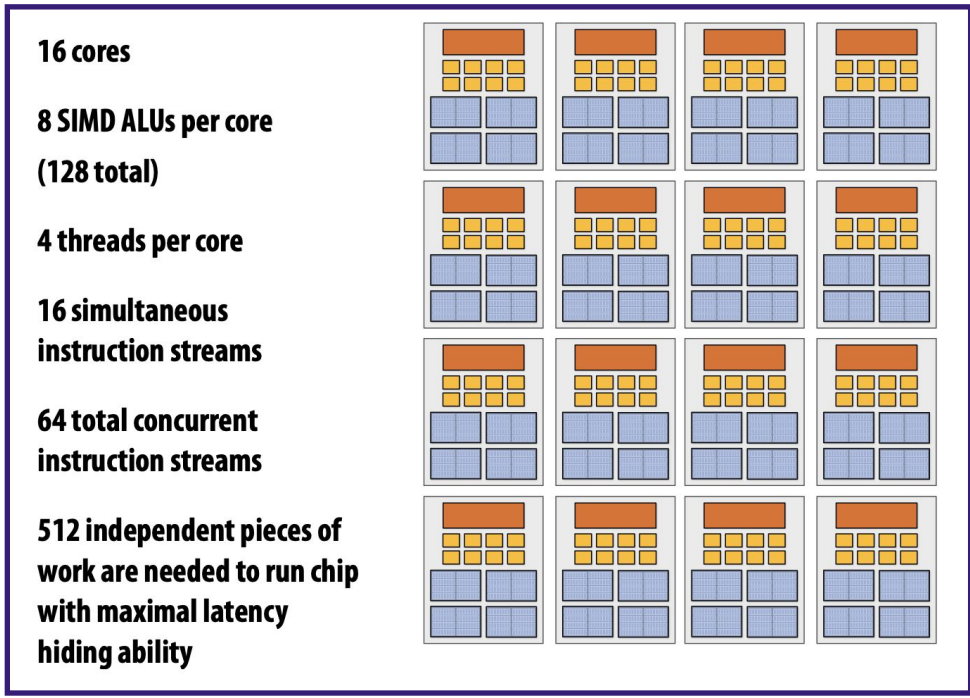


Hardware: Chips

Additional material!

Fictitious multicore chip

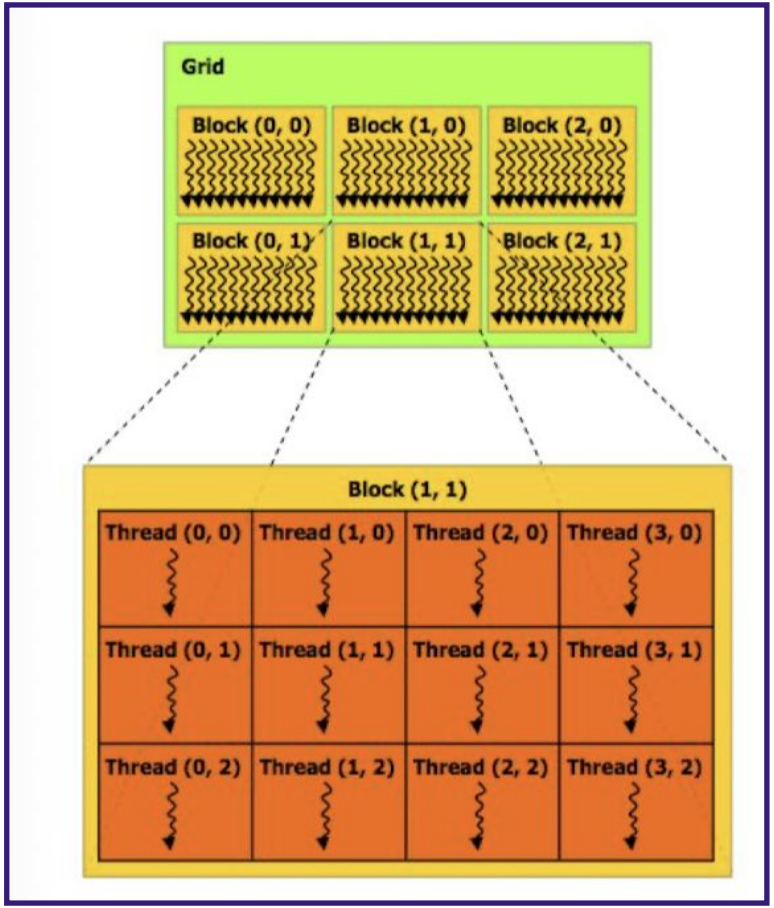
- 4 warp/core x 8 thread/warp x 16 cores/chip = **512** pieces of work accessible on the chip
- Only 8 x 16 = 128 threads are actually executing at any one time
  - The other 384 threads are ready to be swapped in when the others are idle



CUDA: Design

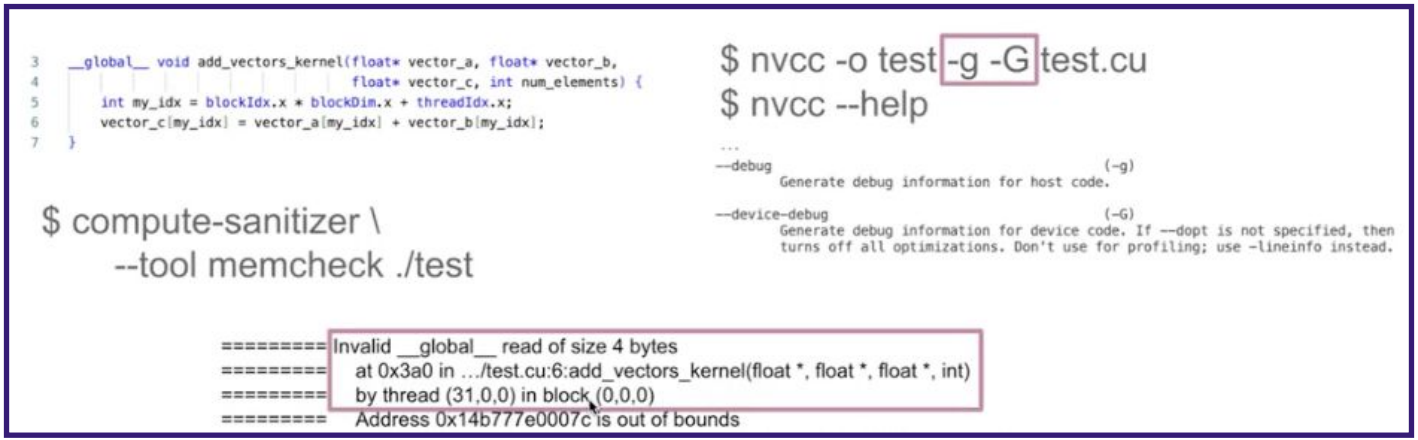
Additional material!

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4,3);
dim3 numBlocks(Nx/threadsPerBlock.x,
Ny/threadsPerBlock.y);

// assume A,B,C are allocated Nx x Ny
float arrays
// this call will launch 72 CUDA
threads: 6 thread blocks of 12 threads
each
TODO matrixAdd(A,B,C){return A+B+C;}
matrixAdd<<<numBlocks,threadsPerBlock
>>>(A,B,C)
```



CUDA: Error Catching

- **Out of bounds**
  - `compute-sanitizer --tool memcheck ./test`
  - Catches OOB memory and tells you exactly which thread/block triggered it



CUDA: Error Catching

- **Wrong Launch Number**
  - Found using `cudaGetLastError()`
  - Kernel launch unsuccessful because it launched with 2024 threads per block when the max number was 1024

```
#include <stdio.h>

__global__ void example_kernel() { return; }

int main() {
    float* example_alloc;
    cudaError_t malloc_return_code = cudaMalloc(&example_alloc, 400);
    if (malloc_return_code != cudaSuccess) {
        printf("%s\n", cudaGetErrorString(malloc_return_code));
    } else {
        printf("Successful malloc.\n");
    }

    example_kernel<<<1, 2048>>>();
    cudaError_t kernel_return_code = cudaGetLastError();
    if (kernel_return_code != cudaSuccess) {
        printf("%s\n", cudaGetErrorString(kernel_return_code));
    } else {
        printf("Successful kernel launch.\n");
    }

    return 0;
}
```

```
$ nvcc -o scratch scratch.cu && ./scratch
Successful malloc.
invalid configuration argument
```

CUDA: Error Catching

- **The problem:** `example_kernel<<<1, 2048>>>();`
  - Can’t launch more than 1024 threads in a block
- **Wrong Launch Number Fix**

CUDA: Error Catching

- **The problem:** `example_kernel<<<1, 2048>>>();`
  - Can't launch more than 1024 threads in a block
- **Wrong Launch Number Fix**
  - `example_kernel<<<1, 1024>>>();`
    - Max allowed in one block
  - `example_kernel<<<2, 1024>>>();`
    - Split the same 2048 threads across 2 blocks

Additional Resources + Sources

- Does PSU Affect FPS? Impact on Gaming Performance
- Accelerated Ray Tracing in One Weekend in CUDA
- Which hardware and games support ray tracing?
- NVIDIA: Real Time Ray Tracing
- Build AI-Powered Games with NVIDIA
- Which GPUs Support Ray Tracing: A Comprehensive Guide
- Stanford CS149: Parallel Computing
- Geeks for Geeks: Clock Speed

Attendance:
https://tinyurl.com/58meh7w8