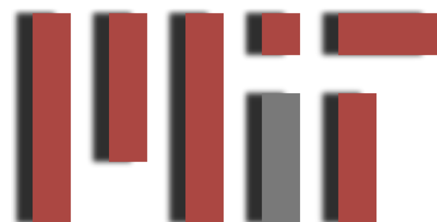
Unlocking Brain-Inspired Computer Vision

Nicolas Pinto

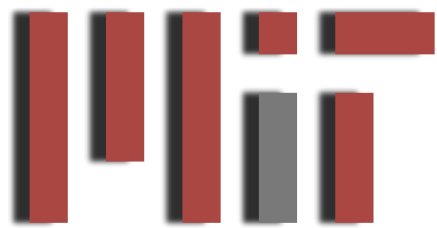


The Rowland Institute at Harvard
HARVARD UNIVERSITY

Unlocking Biologically-Inspired Computer Vision:
a High-Throughput Approach

BU Edition

Nicolas Pinto, David Cox and James DiCarlo
Boston University | November, 2009

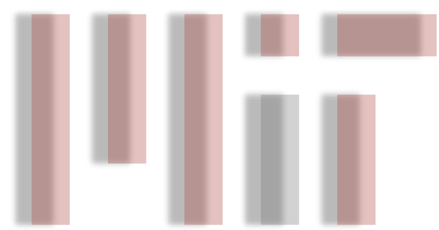


The Rowland Institute at Harvard
HARVARD UNIVERSITY

Unlockin ... ed

Comput ...

a High-T ... n



... tute at Harvard

... SITY

Unlocking Biologically-Inspired Computer Vision: a High-Throughput Approach

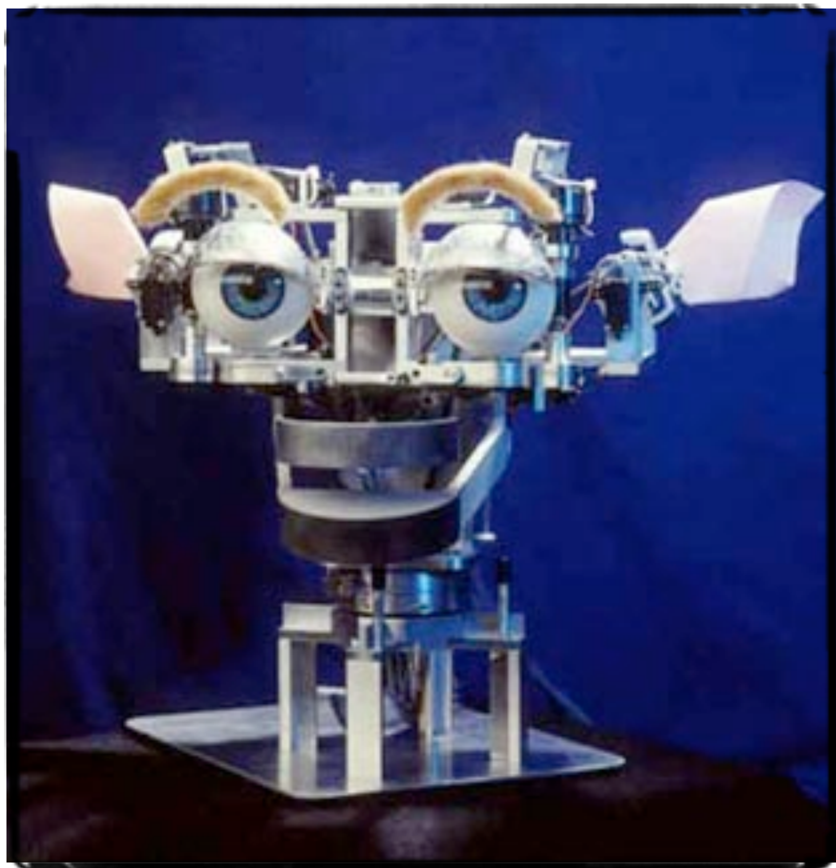
=

BRAIN

(NEUROSCIENCES)

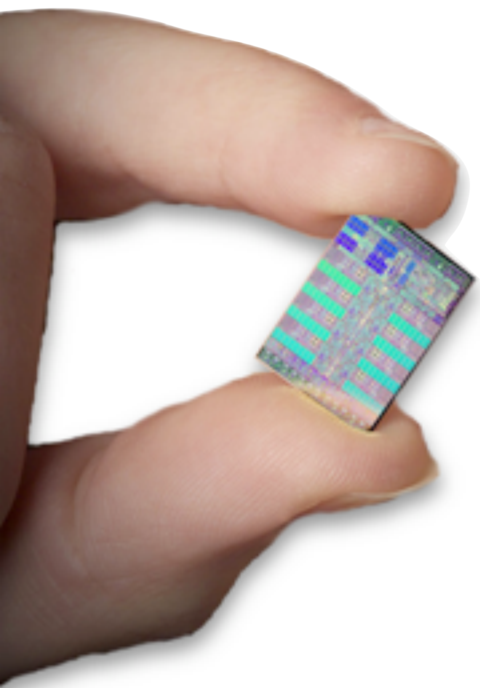
Unlocking Biologically-Inspired Computer Vision: a High-Throughput Approach

= A.I.



Unlocking Biologically-Inspired Computer Vision: a High-Throughput Approach

= GPU



“”

Quote to remember...

Friend: **So, what are you studying for your PhD?**

Me: **I study biological and artificial vision.**

Friend: **What?!? But vision is super easy!**

The Problem:

Visual Object Recognition

- *Fast*
- *Accurate*
- *Tolerant to variation*
- *Effortless*
- *Critical to survival*

(for primates)

// the world is **3D** but the retina is **2D**

// the curse of **dimensionality**

// considerable **image variation**

image variation!

do you recognize me ?

image variation!

011	011	011	011	010	009	009	008	008	008	008	008	007	007	008	007	006	007	006	007	009	010	016	017	016	016	018	022	024	026	027	022
011	010	010	010	009	009	008	008	008	008	008	008	008	008	008	006	006	006	004	006	009	010	013	016	015	018	029	035	035	034	031	022
012	011	011	010	010	009	008	008	008	009	008	008	008	008	008	006	006	007	004	006	009	009	013	016	015	018	030	035	033	034	032	024
011	011	011	010	010	008	008	009	008	008	009	008	008	007	008	008	007	008	005	007	009	008	012	016	015	016	027	037	038	036	032	025
013	012	010	010	009	008	006	007	008	009	009	008	008	008	008	008	008	008	008	007	009	008	011	015	014	016	026	037	037	033	029	025
011	011	011	010	008	007	005	006	009	009	009	007	006	007	007	008	008	008	007	007	008	008	013	016	015	015	023	034	034	032	031	029
010	010	010	009	008	006	004	005	009	009	008	006	005	007	008	008	008	008	008	008	008	009	013	016	015	015	022	036	040	038	036	034
010	010	010	009	008	006	005	006	009	010	008	007	007	008	007	008	008	008	008	008	008	010	013	016	015	014	021	037	040	037	035	033
010	010	010	009	008	007	007	010	009	009	008	008	008	007	006	007	008	008	007	008	008	010	014	016	015	015	019	030	034	033	032	030
011	010	010	009	009	007	007	009	009	009	009	008	007	006	006	007	008	008	008	007	007	012	016	016	015	015	017	027	033	033	033	034
013	011	010	009	009	007	008	011	011	011	011	009	008	008	007	007	008	008	008	008	008	014	018	016	016	015	017	026	035	034	036	037
017	013	012	014	011	006	008	011	012	014	013	010	009	010	007	006	008	008	008	008	008	015	017	016	015	016	017	025	035	035	036	035
021	017	015	017	013	007	007	009	012	013	012	009	008	006	005	005	006	006	006	007	009	015	018	017	016	015	016	023	033	033	033	041
021	018	015	017	013	008	010	011	012	013	013	009	009	007	006	007	007	007	008	010	013	018	020	019	016	016	016	022	033	041	056	074
020	016	013	014	013	012	013	014	014	014	014	012	011	009	008	011	014	015	016	016	017	020	022	020	016	015	014	022	045	056	062	066
019	017	012	013	015	017	019	018	015	014	013	012	011	010	009	013	016	016	018	017	016	018	021	020	018	017	015	032	049	053	056	053
022	018	010	011	016	021	027	024	016	014	013	013	011	009	008	014	017	018	018	018	018	019	023	021	019	017	025	038	041	041	036	037
023	018	012	016	017	018	023	023	017	016	017	018	015	012	012	015	017	022	022	022	022	023	023	022	020	018	028	035	036	030	025	028
024	018	014	018	018	015	015	015	015	015	015	020	024	023	021	016	015	017	022	022	023	022	022	022	020	020	025	028	029	026	028	030
026	021	015	017	021	021	018	015	014	015	015	022	028	025	023	017	016	016	021	021	021	021	021	021	021	022	023	024	025	026	029	031
025	023	016	016	023	026	023	019	018	019	019	023	028	027	019	014	015	017	019	028	037	028	021	022	022	024	023	024	023	026	031	032
025	021	016	014	017	021	020	020	025	028	031	047	073	081	067	043	023	014	038	075	087	072	033	019	024	023	022	025	025	029	032	032
021	020	014	014	020	018	016	018	022	028	054	093	123	138	133	112	082	075	100	121	124	109	067	023	022	020	021	026	027	032	032	032
023	020	010	010	022	021	017	019	022	033	081	120	132	141	142	135	127	127	128	125	123	125	097	040	019	020	021	[illegible]	029	033	033	030
022	018	011	014	025	029	028	029	036	052	096	127	126	129	130	126	125	123	120	116	118	127	111	053	[illegible]	[illegible]	[illegible]	[illegible]	032	035	034	032
023	019	015	016	025	033	034	035	044	058	097	124	123	123	123	117	116	115	118	115	106	113	114	071	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	035	033	033
028	028	023	020	028	034	035	040	046	064	103	121	113	119	125	117	112	111	118	112	079	070	095	081	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	034	032	030
033	035	033	034	037	038	040	044	056	086	119	109	095	117	127	117	110	108	116	109	070	034	052	062	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	035	032	029
033	033	034	034	035	035	036	038	044	075	098	082	089	122	124	115	110	109	115	107	073	034	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	035	030	027
027	028	028	028	027	029	030	032	035	041	046	059	101	127	124	116	109	111	117	108	073	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	034	030	025
032	032	032	[illegible]	032	032	[illegible]	034	036	037	038	069	113	130	124	119	115	114	120	110	081	051	021	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	022
037	039	038	038	039	038	037	037	039	040	044	081	120	131	126	121	119	118	119	109	092	[illegible]	030	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

do you recognize me ?

the brain!

~50% of that is for vision!

you learned it...

Need for speed

Hardware

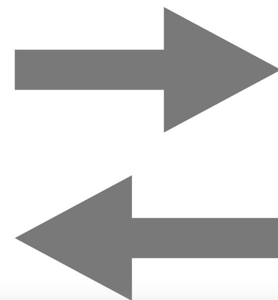
Software

Science

The Approach:
Reverse Engineering the Brain

REVERSE

Study
Natural System

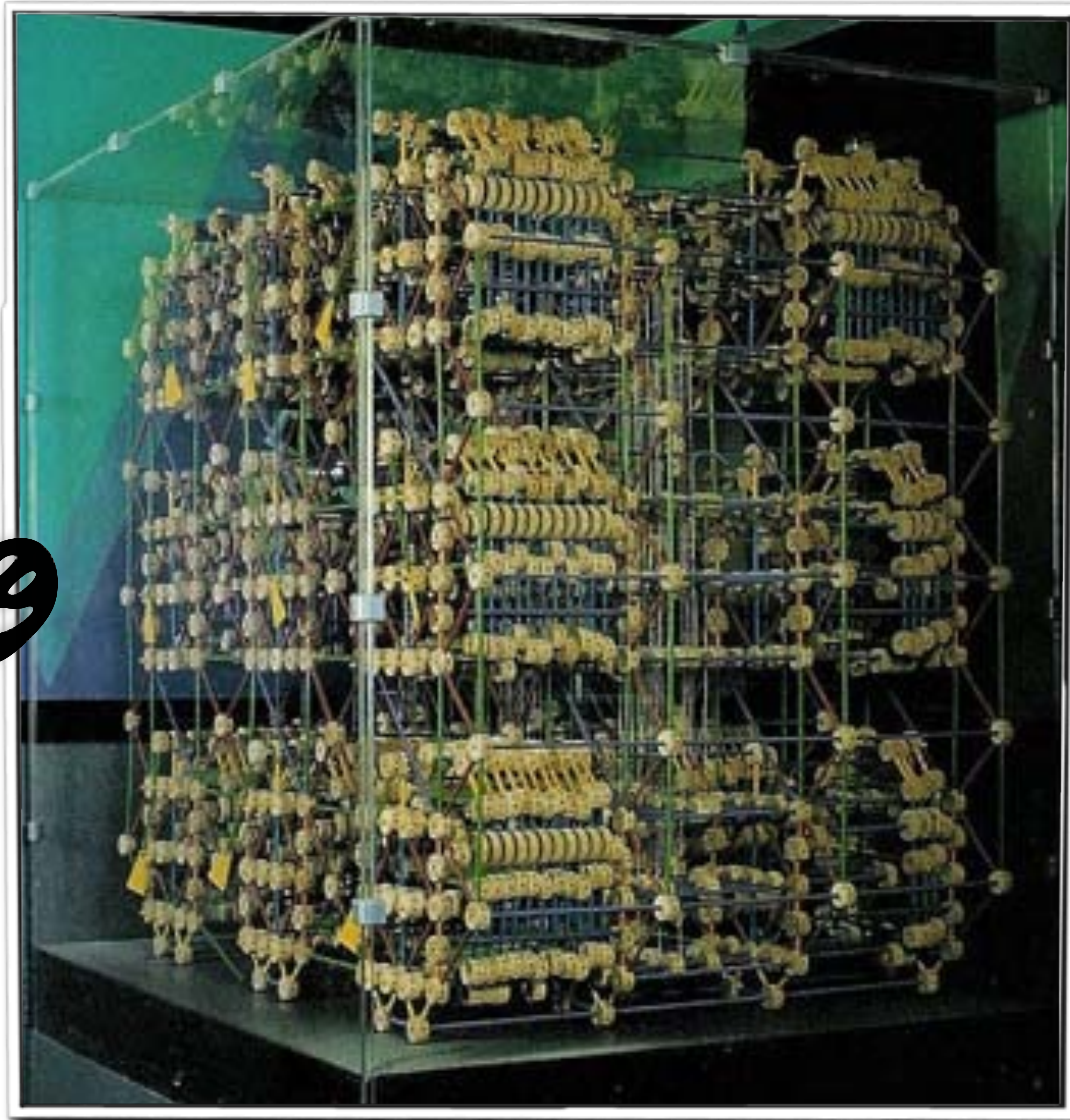


FORWARD

Build
Artificial System

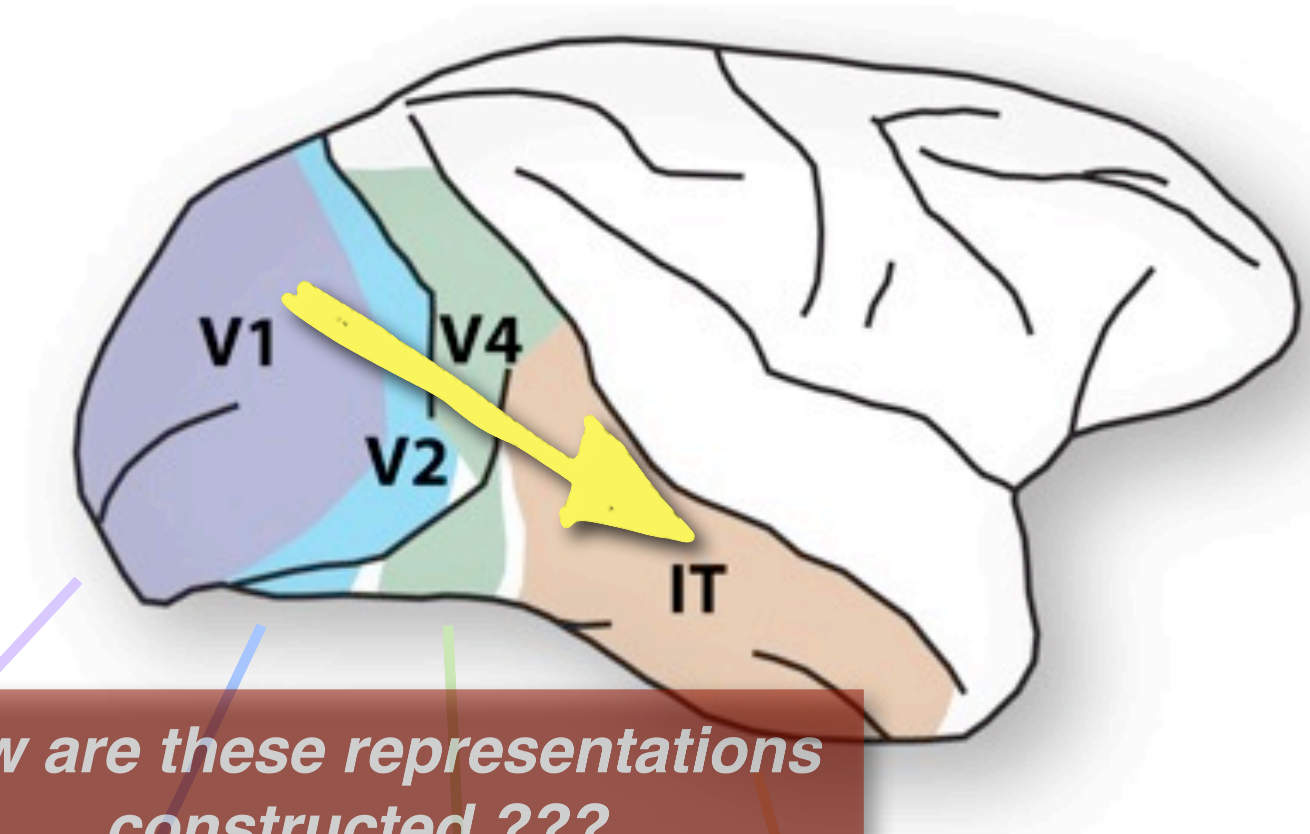
Reverse Engineering ...

What is this doing? ☹

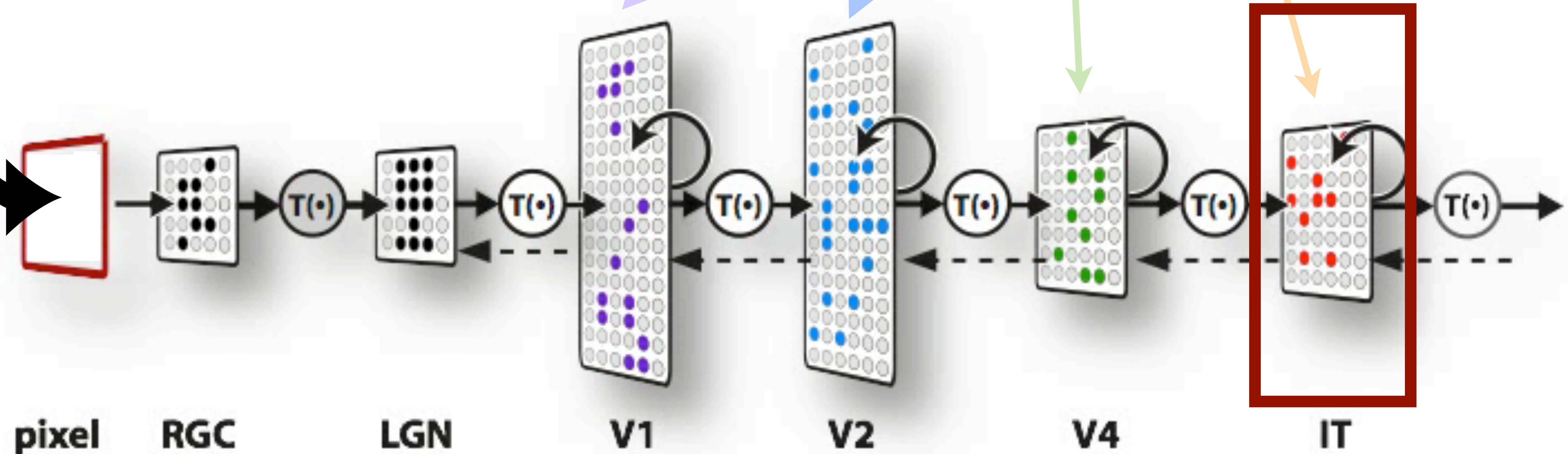


Reverse Engineering **the Brain!**

The Ventral Visual Stream



How are these representations constructed ???



IT Cortex can do object recognition

Classification performance (% correct)

100
80
60
40
20
0

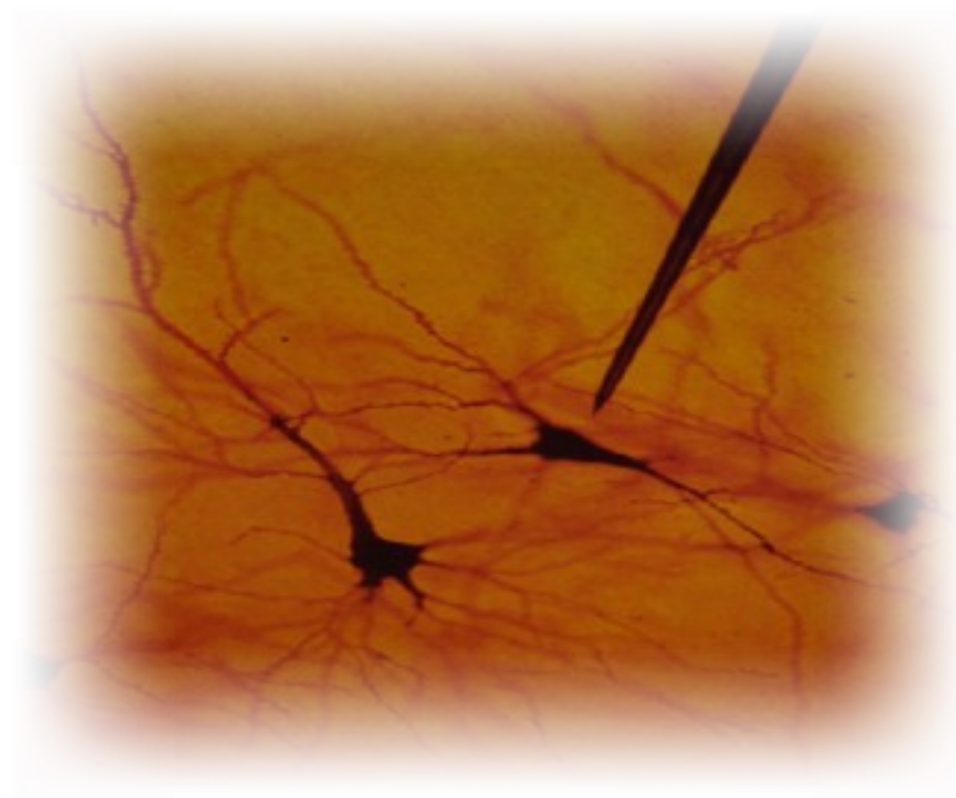
I

Size: 3.4° center | 3.4° center | 1.7° center | 6.8° center | 3.4° 2° horz. | 3.4° 4° horz.

Position:

TRAIN

TEST



Hung, Kreiman*, Poggio and DiCarlo,* ***Science*** *(2005)*

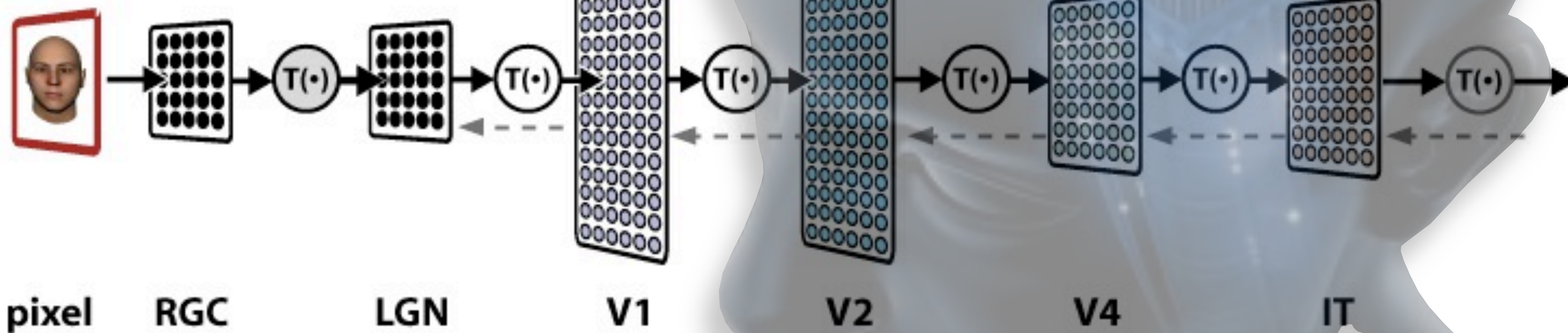
Visual Cortex

brain = 20 petaflops !

V1

IT

10 mm



The need for **speed**

- **billions** of neurons and synapses
- **large-scale** natural evolution (“high-throughput screening” of neural architectures)
- **hours** of unsupervised learning experience
- faithful reproduction of other models (i.e. blend **many highly tuned** techniques)

Our strategy

Capitalizing on non-scientific high-tech markets and their $billions of R&D...

- **Gaming:** GPUs, PlayStation 3 (CellBE)
- **Web 2.0:** Cloud Computing (Amazon, Google)

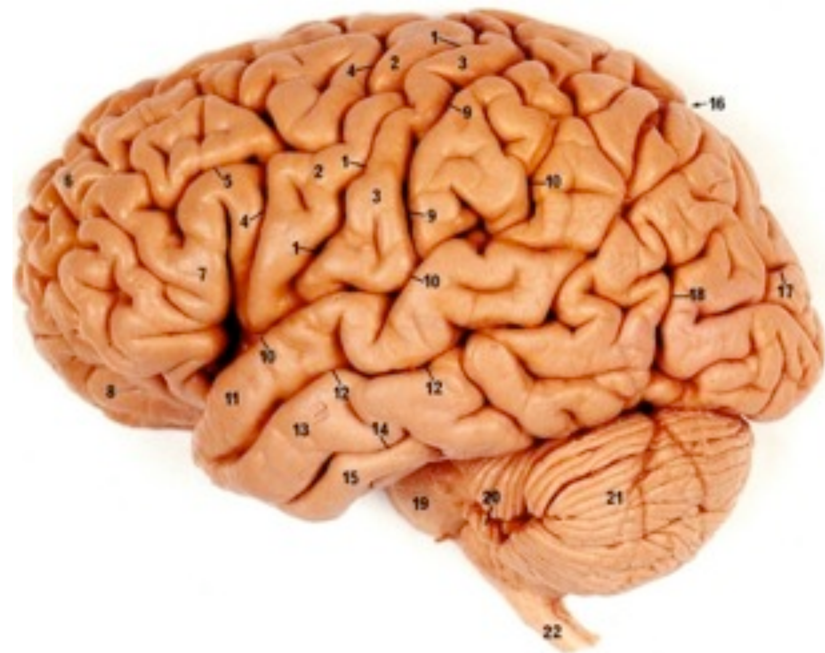
Need for speed

Hardware

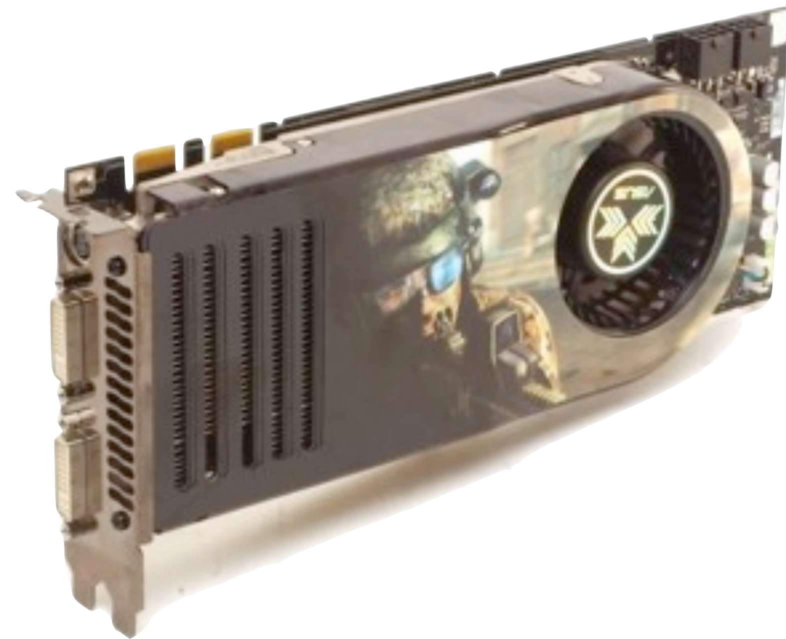
Software

Science

A Match Made in Heaven
Brains are parallel, **GPUs** are parallel



≈



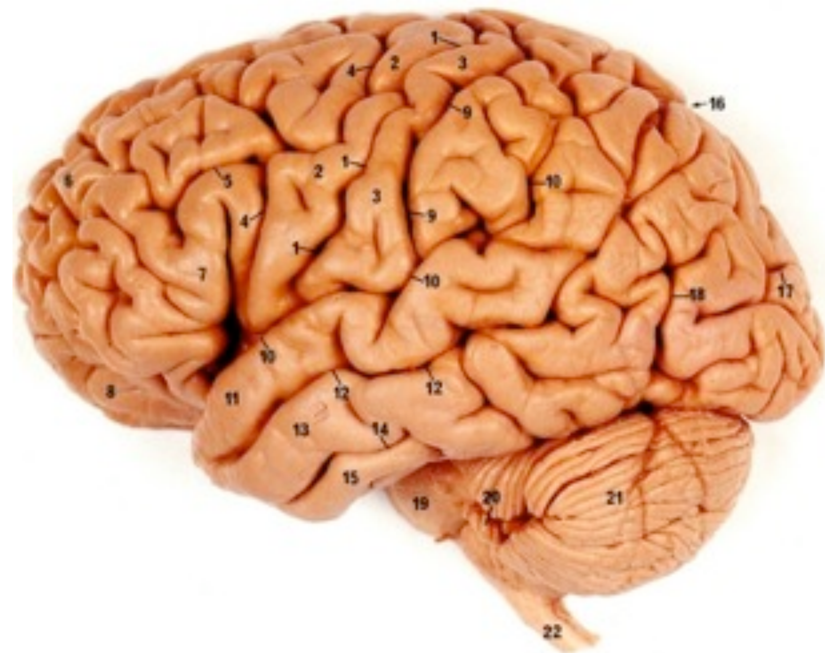
Multiple scales of parallelism:

"Embarrasingly" parallel: video frames, regions

Fine-grained: independent "neurons," operating on overlapping inputs

A Match Made in Heaven

Images In, Images Out



≈

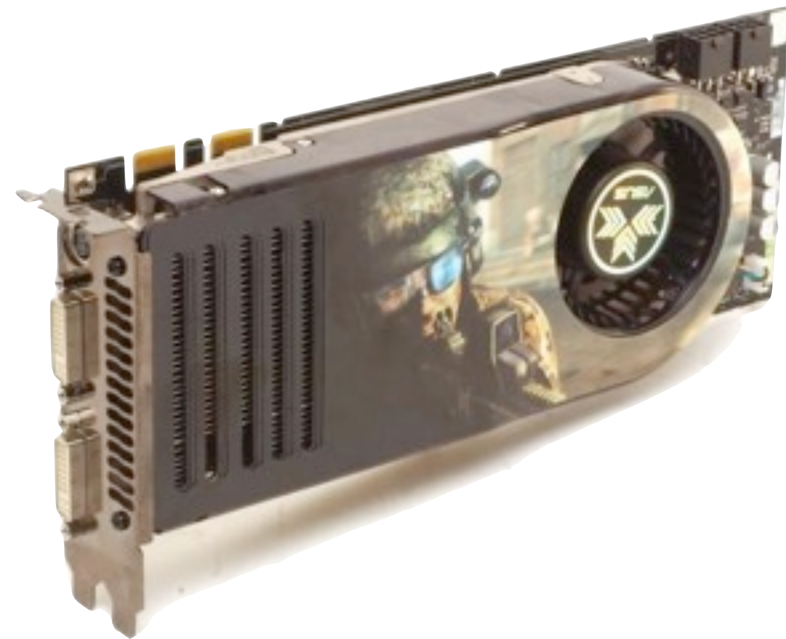


Image processing particularly well-suited

Excellent Arithmetic Intensity: very natural to load image patches into shared memory

Data: 2D / 3D locality

GPUs (since 2006)

7800 GTX (2006)	Monster16GPU (2008)	Tesla Cluster (2009)
OpenGL/Cg	*CUDA*	*CUDA/OpenCL*
C++/Python	*Python*	*Python*

Build your own!
Our 16-GPU Monster-Class Supercomputer
the world's most compact (18"x18"x18") and inexpensive ($3000) supercomputer

Cell Broadband Engine (since 2007)

Teraflop Playstation3 clusters:

DiCarlo Lab / MIT

Cox Lab / Harvard

Amazon Cloud Computing (since 2008)

Cloud Computing Services

Integration

Elastic Compute Cloud (EC2)

Machine Images

On-Demand Instances

Support Services

Simple Storage Service (S3)

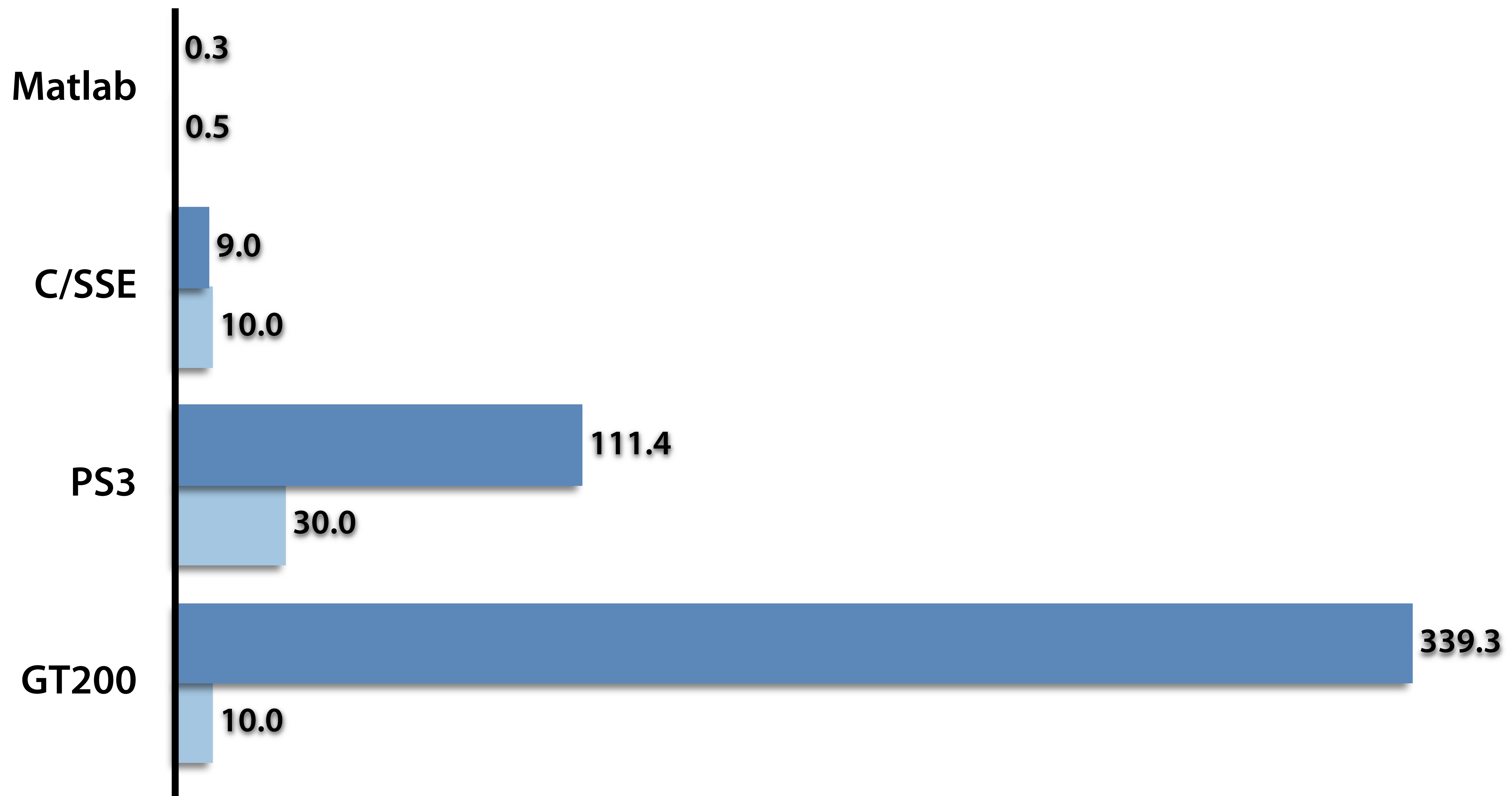
SimpleDB

CloudFront

Simple Queue Service (SQS)

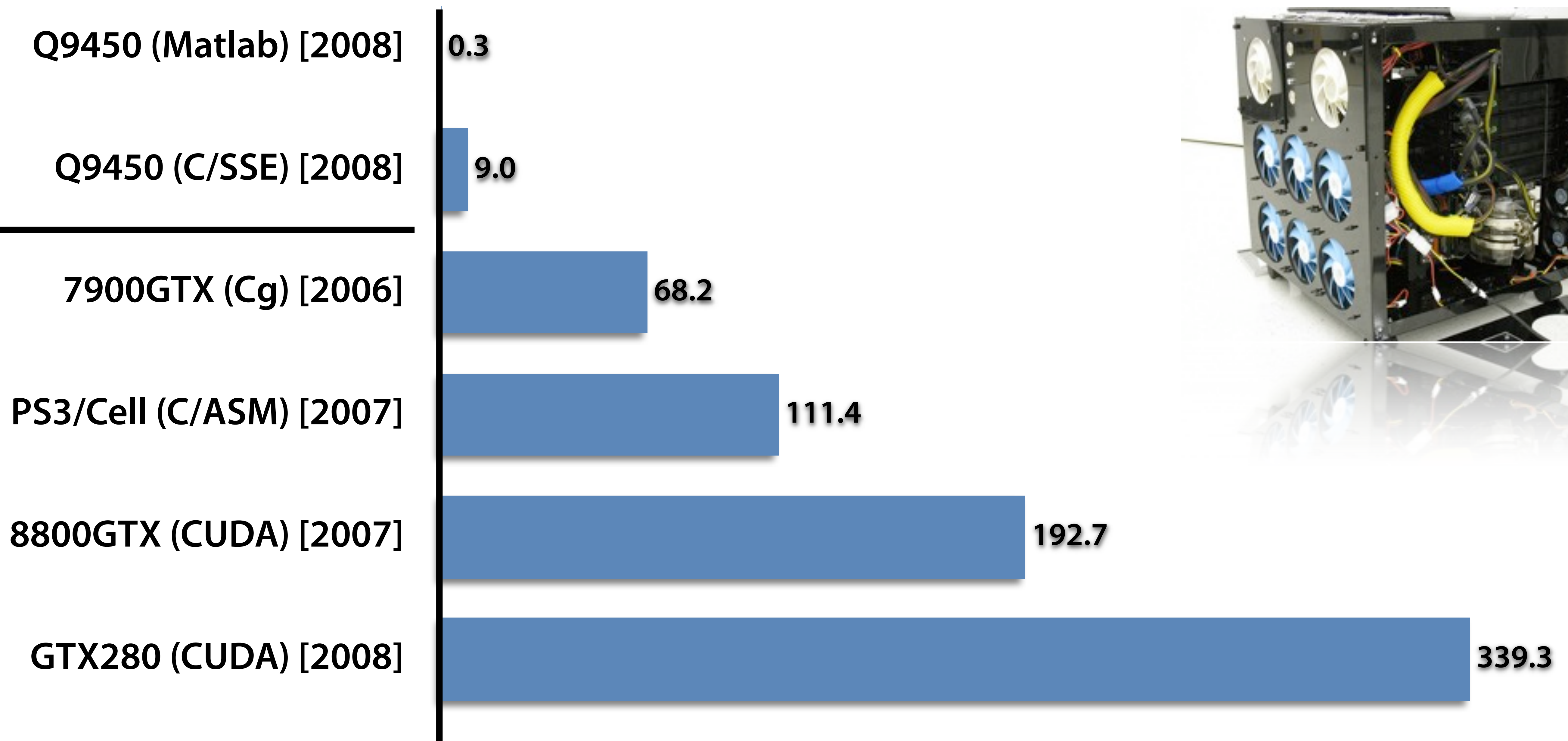
Some numbers...

3D Filterbank Convolution



Some numbers...

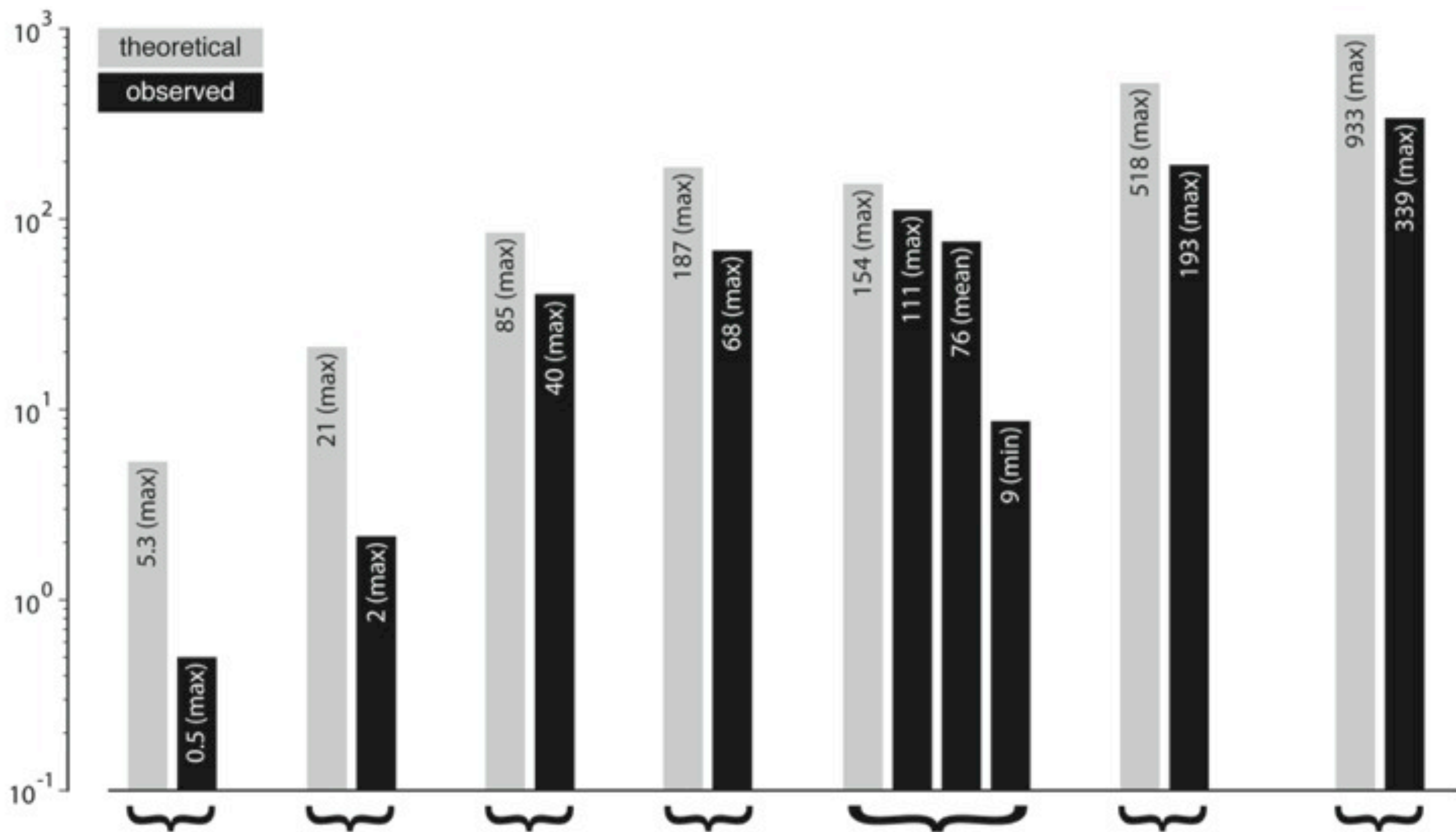
3D Filterbank Convolution



Some numbers...

3D Filterbank Convolution

Hardware	CPUs			GPUs			
Manufacturer	Intel	Intel	Intel	NVIDIA	Sony, IBM, Toshiba	NVIDIA	NVIDIA
Model	Q9450	Q9450	Q9450	7900 GTX	PlayStation 3	8800 GTX	GTX 280
# cores used	1	4	4	4x96	2+6	4x128	4x240
Implementation	MATLAB	MATLAB	SSE2	Cg	Cell SDK	CUDA	CUDA
Year	2008	2008	2008	2006	2007	2007	2008
Performance / Cost							
Full System Cost (approx.)	$1,500**	$2,700**	$1,000	$3,000*	$400	$3,000*	$3,000*
Relative Speedup	**1x**	**4x**	**80x**	**544x**	**222x**	**1544x**	**2712x**
Relative Perf. / $	**1x**	**2x**	**120x**	**272x**	**833x**	**772x**	**1356x**



Hardware							
Manufacturer	Intel	Intel	Intel	NVIDIA	Sony, IBM, Toshiba	NVIDIA	NVIDIA
Model	Q9450	Q9450	Q9450	7900 GTX	PlayStation 3 (Cell)	8800 GTX	GTX 280
# cores used	1	4	4	96	8 (2+6)	128	240
Software							
Languages	MATLAB / C	MATLAB / C	C	Python / C++	Python / C	Python / C	Python / C
Extensions / Libraries	MEX	MEX	SSE2 / pthread	OpenGL / Cg	Cell SDK	PyCUDA	PyCUDA
Year of implementation	2008	2008	2008	2006	2007	2007	2008
Performance / Cost							
GFLOPS (max)	0.5	2	40	68-272*	111	193-772*	339-1356*
Full System Cost (approx.)	$1,500**	$2,700**	$1,000	$1,500-$3,000*	$400	$1,500-$3,000*	$1,500-$3,000*
$ / GFLOPS	3000	1350	25	22-11*	4	8-4*	4-2*
Relative Speedup	1	4	80	136-544*	222	386-1544*	678-2712*
Relative GFLOPS / $	1	2	120	136-272*	833	386-772*	678-1356*

Need for speed
Hardware
Software
Science

What do we all want?

- Ease of use
- **Maximum raw speed**
- Ease of extension
- Hardware “agnostic”

A little story

You just finished your code...

1. You run it on one image: it works!
2. You adjust your parameters: it’s slow!
3. Your optimize your code: it’s fast now!
4. You run it on another image: it’s slow now!
5. You repeat or you stop...

Here are the keys to Easy-High-Performance !

Meta-programming?

Meta-programming !

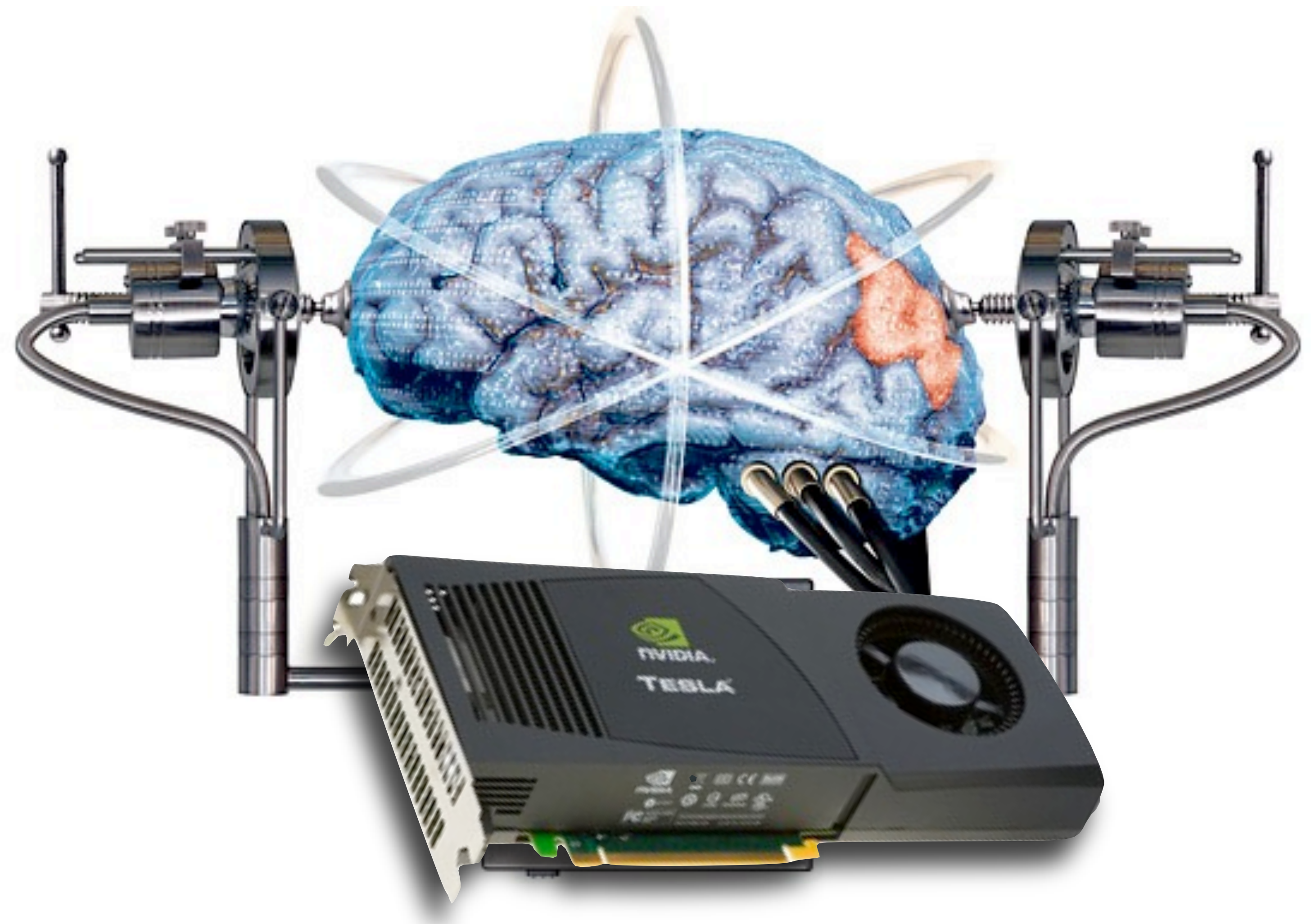
Leave the **grunt-programming** to the computer (i.e. auto-tuning like ATLAS or FFTW)

- Dynamically compile **specialized versions** of the same kernel for different conditions (~Just-in-Time Compilation (JIT))
- **Smooth** syntactic ugliness: unroll loops, index un-indexable registers
- **Dynamic**, empirical run-time **tuning**

Meta-programming!

“Instrumentalize” your solutions:

- Block size
- Work size
- Loop unrolling
- Pre–fetching
- Spilling
- **etc.**



Meta-programming!

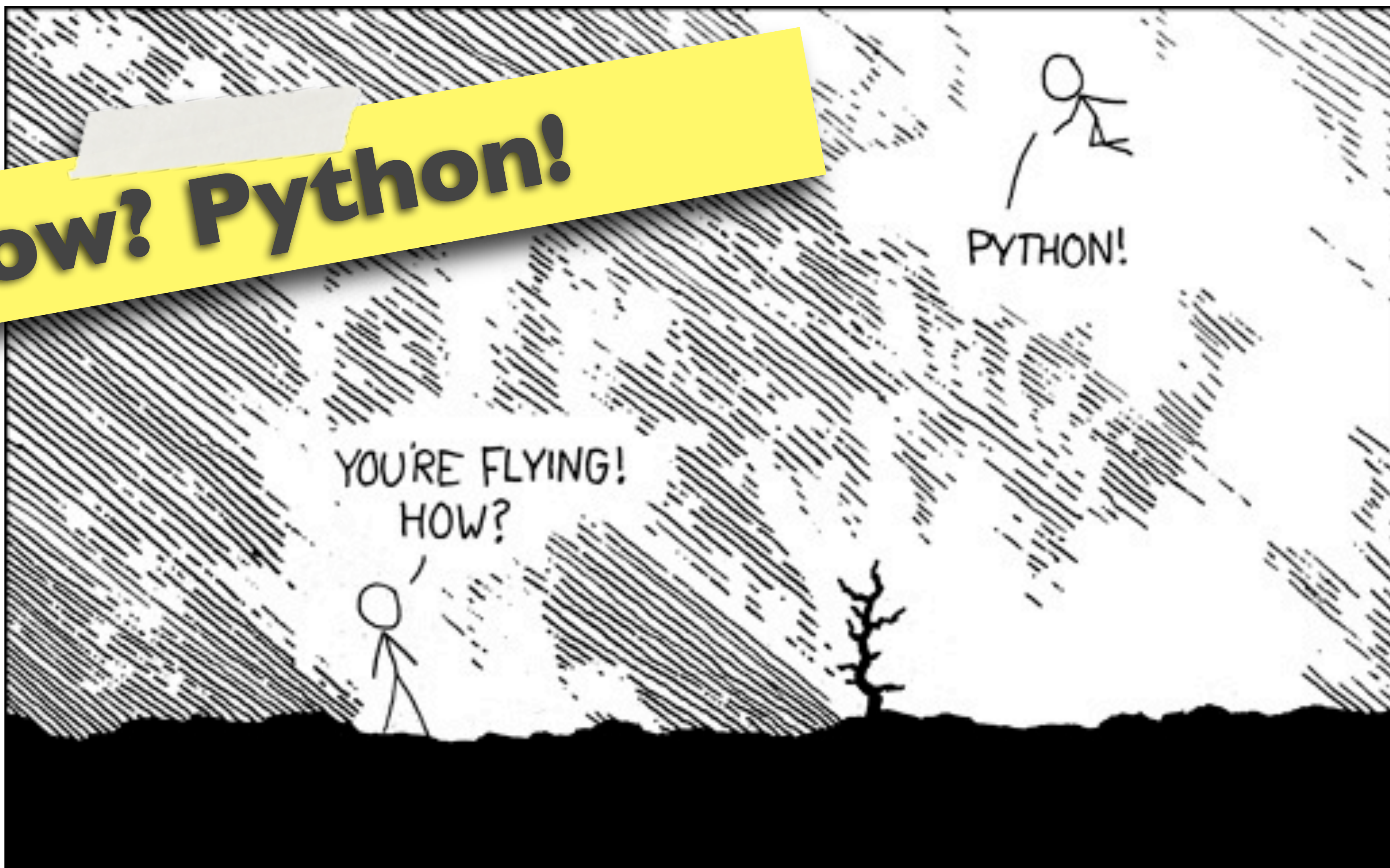
Let the computer find the **optimal code**:

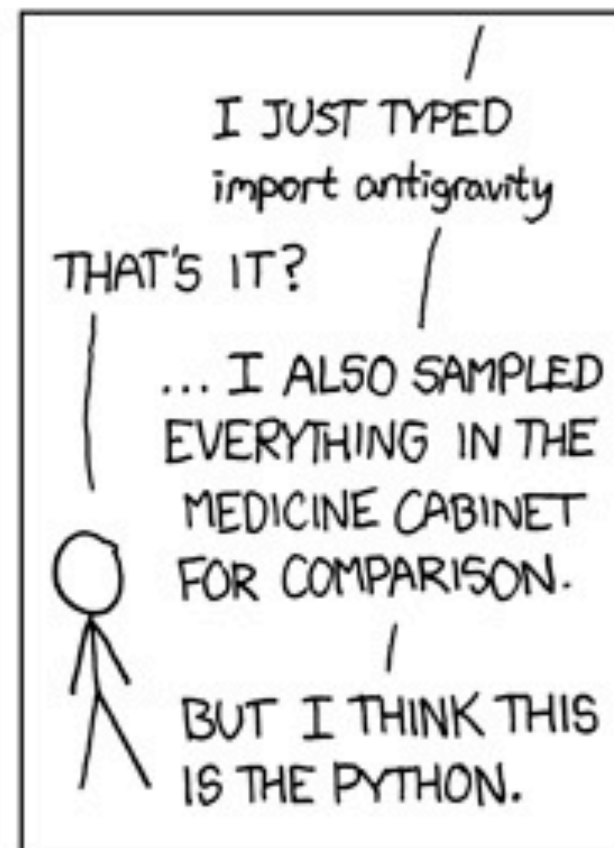
- brute–force search with a **global objective**
- machine–learning approach with **local objectives** and **hidden variables** (advanced)
  - eg. PyCuda makes this easy:
    - Access properties of compiled code: `func.{registers,lmem,smem}`
    - Exact GPU timing via events
    - Can calculate hardware-dependent MP occupancy

How ?

Our mantra: always use the right tool !

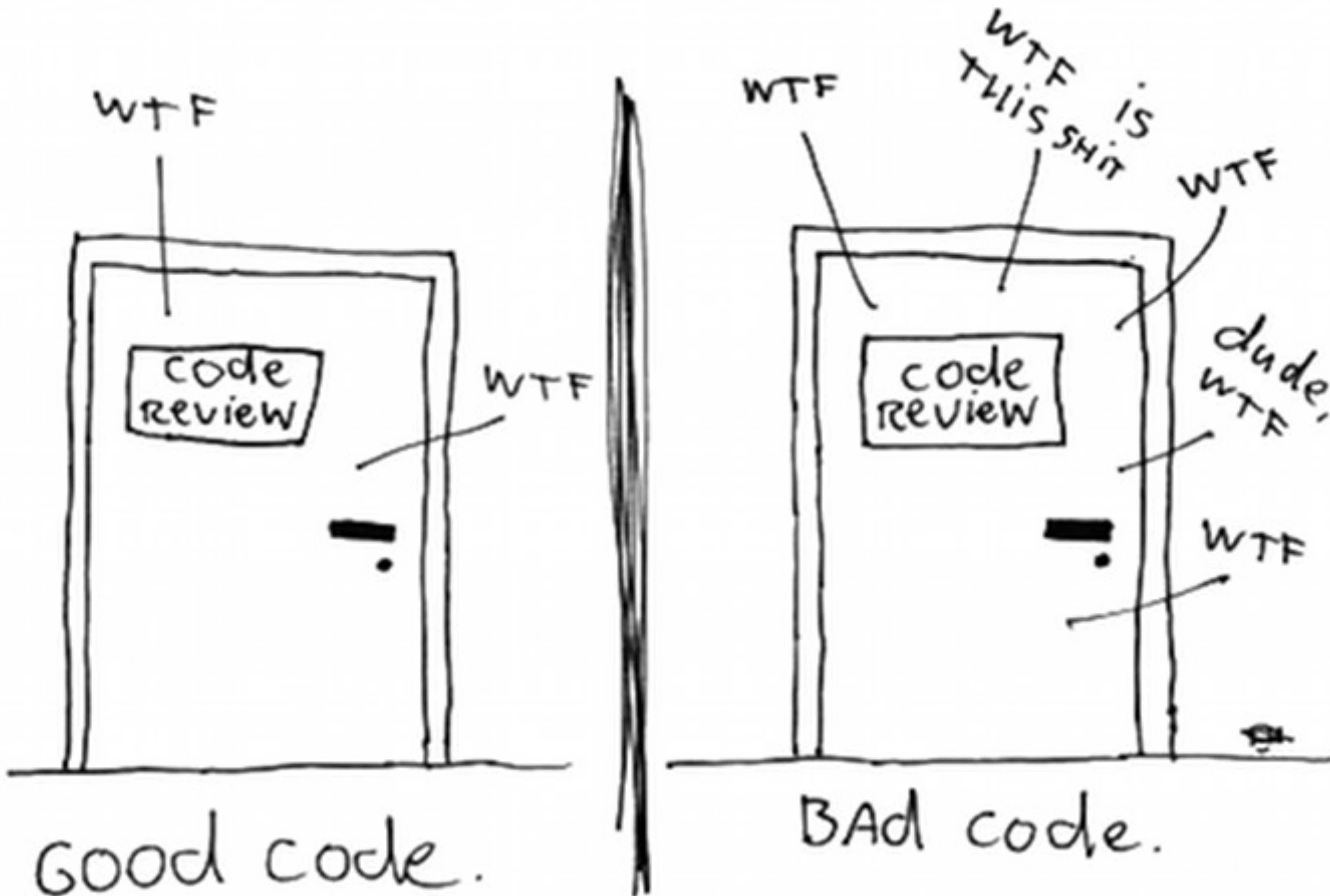
How? Python!





Meta-programming requires **careful engineering**

Meta-programming requires **careful engineering**



```
texture<float4, 1, cudaReadModeElementType> tex_float4;
__constant__ float constant[$FILTER_D][$FILTER_W][$N_FILTERS];

#define IMUL(a, b) __mul24(a, b)
extern "C" {

#for j in xrange($FILTER_H)

  __global__ void convolve_beta_j${j}(float4 *input, float4 *output)
  {

#set INPUT_BLOCK_W = $BLOCK_W+$FILTER_W-1
    __shared__ float shared_in[$INPUT_BLOCK_W][4+1];

    // -- input/output offsets
    const uint in_idx = (blockIdx.y+$j)*INPUT_W + blockIdx.x*blockDim.x + threadIdx.x;
    const uint out_idx = blockIdx.y*OUTPUT_W + blockIdx.x*blockDim.x + threadIdx.x;
    float4 input_v4;

    // -- load input to shared memory
#for i in xrange($LOAD_ITERATIONS)
#if $i==($LOAD_ITERATIONS-1)
    if((threadIdx.x+$BLOCK_W*$i)<$INPUT_BLOCK_W)
#end if
      {
    input_v4 = tex1Dfetch(tex_float4, in_idx+$BLOCK_W*$i);
    shared_in[threadIdx.x+$BLOCK_W*$i][0] = input_v4.x;
    shared_in[threadIdx.x+$BLOCK_W*$i][1] = input_v4.y;
    shared_in[threadIdx.x+$BLOCK_W*$i][2] = input_v4.z;
```

conv_kernel_template.cu

```
texture<float4, 1, cudaReadModeElementType> tex_float4;
__constant__ float constant[$FILTER_D][$FILTER_W]
[$N_FILTERS];

#define IMUL(a, b) __mul24(a, b)
extern "C" {

#for j in xrange($FILTER_H)

  __global__ void convolve_beta_j${j}(float4 *input, float4
*output)
  {

#set INPUT_BLOCK_W = $BLOCK_W+$FILTER_W-1
    __shared__ float shared_in[$INPUT_BLOCK_W][4+1];

    // -- input/output offsets
    const uint in_idx = (blockIdx.y+$j)*INPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    const uint out_idx = blockIdx.y*OUTPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    float4 input_v4;

    // -- load input to shared memory
#for i in xrange($LOAD_ITERATIONS)
#if $i==($LOAD_ITERATIONS-1)
    if((threadIdx.x+$BLOCK_W*$i)<$INPUT_BLOCK_W)
#end if
      {
        input_v4 = tex1Dfetch(tex_float4, in_idx+$BLOCK_W*
$i);
        shared_in[threadIdx.x+$BLOCK_W*$i][0] = input_v4.x;
        shared_in[threadIdx.x+$BLOCK_W*$i][1] = input_v4.y;
        shared_in[threadIdx.x+$BLOCK_W*$i][2] = input_v4.z;
        shared_in[threadIdx.x+$BLOCK_W*$i][3] = input_v4.w;
      }
#end for
```

conv_kernel_4x4x4.cu

```
#include <stdio.h>

texture<float4, 1, cudaReadModeElementType> tex_float4;
__constant__ float constant[4][4][4];

#define IMUL(a, b) __mul24(a, b)
extern "C" {


 __global__ void convolve_beta_j0(float4 *input, float4 *output)
 {

   __shared__ float shared_in[131][4+1];

   // -- input/output offsets
   const uint in_idx = (blockIdx.y+0)*INPUT_W + blockIdx.x*blockDim.x + threadIdx.x;
   const uint out_idx = blockIdx.y*OUTPUT_W + blockIdx.x*blockDim.x + threadIdx.x;
   float4 input_v4;

   // -- load input to shared memory
    {
          input_v4 = tex1Dfetch(tex_float4, in_idx+128*0);
          shared_in[threadIdx.x+128*0][0] = input_v4.x;
          shared_in[threadIdx.x+128*0][1] = input_v4.y;
          shared_in[threadIdx.x+128*0][2] = input_v4.z;
          shared_in[threadIdx.x+128*0][3] = input_v4.w;
    }
   if((threadIdx.x+128*1)<131)
    {
          input_v4 = tex1Dfetch(tex_float4, in_idx+128*1);
          shared_in[threadIdx.x+128*1][0] = input_v4.x;
          shared_in[threadIdx.x+128*1][1] = input_v4.y;
          shared_in[threadIdx.x+128*1][2] = input_v4.z;
          shared_in[threadIdx.x+128*1][3] = input_v4.w;
    }
   __syncthreads();

   // -- compute dot products
   float v, w;

   float sum0 = 0;
   float sum1 = 0;
   float sum2 = 0;
   float sum3 = 0;

   v = shared_in[threadIdx.x+0][0];
   w = constant[0][0][0];
   sum0 += v*w;
   w = constant[0][0][1];
   sum1 += v*w;
   w = constant[0][0][2];
   sum2 += v*w;
   w = constant[0][0][3];
   sum3 += v*w;
   v = shared_in[threadIdx.x+1][0];
   w = constant[0][1][0];
   sum0 += v*w;
   w = constant[0][1][1];
   sum1 += v*w;
   w = constant[0][1][2];
   sum2 += v*w;
   w = constant[0][1][3];
   sum3 += v*w;
   v = shared_in[threadIdx.x+2][0];
   w = constant[0][2][0];
   sum0 += v*w;
   w = constant[0][2][1];
   sum1 += v*w;
```

conv_kernel_template.cu

```
texture<float4, 1, cudaReadModeElementType> tex_float4;
__constant__ float constant[$FILTER_D][$FILTER_W]
[$N_FILTERS];

#define IMUL(a, b) __mul24(a, b)
extern "C" {

#for j in xrange($FILTER_H)

  __global__ void convolve_beta_j${j}(float4 *input, float4
*output)
  {

#set INPUT_BLOCK_W = $BLOCK_W+$FILTER_W-1
    __shared__ float shared_in[$INPUT_BLOCK_W][4+1];

    // -- input/output offsets
    const uint in_idx = (blockIdx.y+$j)*INPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    const uint out_idx = blockIdx.y*OUTPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    float4 input_v4;

    // -- load input to shared memory
#for i in xrange($LOAD_ITERATIONS)
#if $i==($LOAD_ITERATIONS-1)
    if((threadIdx.x+$BLOCK_W*$i)<$INPUT_BLOCK_W)
#end if
      {
        input_v4 = tex1Dfetch(tex_float4, in_idx+$BLOCK_W*
$i);
        shared_in[threadIdx.x+$BLOCK_W*$i][0] = input_v4.x;
        shared_in[threadIdx.x+$BLOCK_W*$i][1] = input_v4.y;
        shared_in[threadIdx.x+$BLOCK_W*$i][2] = input_v4.z;
        shared_in[threadIdx.x+$BLOCK_W*$i][3] = input_v4.w;
      }
#end for
```

conv_kernel_4x4x4.cu

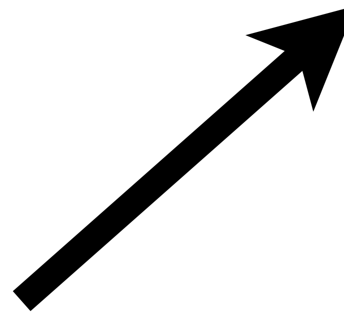
20 kB

conv_kernel_8x8x4.cu

64 kB

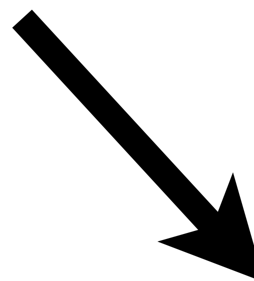
conv_kernel_beta_template.cu

```
texture<float4, 1, cudaReadModeElementType> tex_float4;
__constant__ float constant[$FILTER_D][$FILTER_W]
[$N_FILTERS];

#define IMUL(a, b) __mul24(a, b)
extern "C" {

#for j in xrange($FILTER_H)

  __global__ void convolve_beta_j${j}(float4 *input, float4
*output)
  {

#set INPUT_BLOCK_W = $BLOCK_W+$FILTER_W-1
    __shared__ float shared_in[$INPUT_BLOCK_W][4+1];

    // -- input/output offsets
    const uint in_idx = (blockIdx.y+$j)*INPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    const uint out_idx = blockIdx.y*OUTPUT_W +
blockIdx.x*blockDim.x + threadIdx.x;
    float4 input_v4;

    // -- load input to shared memory
#for i in xrange($LOAD_ITERATIONS)
#if $i==($LOAD_ITERATIONS-1)
    if((threadIdx.x+$BLOCK_W*$i)<$INPUT_BLOCK_W)
#end if
      {
        input_v4 = tex1Dfetch(tex_float4, in_idx+$BLOCK_W*
$i);
        shared_in[threadIdx.x+$BLOCK_W*$i][0] = input_v4.x;
        shared_in[threadIdx.x+$BLOCK_W*$i][1] = input_v4.y;
        shared_in[threadIdx.x+$BLOCK_W*$i][2] = input_v4.z;
        shared_in[threadIdx.x+$BLOCK_W*$i][3] = input_v4.w;
      }
#end for
```



version A

```
...
mad.rn.f32 $r4, s[$ofs3+0x0000], $r4, $r1
mov.b32 $r1, c0[$ofs2+0x0008]
mad.rn.f32 $r4, s[$ofs3+0x0008], $r1, $r4
mov.b32 $r1, c0[$ofs2+0x000c]
mad.rn.f32 $r4, s[$ofs3+0x000c], $r1, $r4
mov.b32 $r1, c0[$ofs2+0x0010]
mad.rn.f32 $r4, s[$ofs3+0x0010], $r1, $r4
...
```

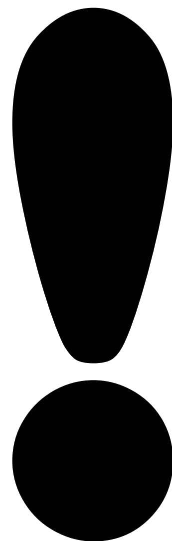


version B

```
...
mad.rn.f32 $r1, s[$ofs1+0x007c], c0[$ofs1+0x0078], $r1
mad.rn.f32 $r1, s[$ofs2+0x0000], c0[$ofs2+0x007c], $r1
mad.rn.f32 $r1, s[$ofs2+0x0008], c0[$ofs2+0x0080], $r1
mad.rn.f32 $r1, s[$ofs2+0x000c], c0[$ofs2+0x0084], $r1
mad.rn.f32 $r1, s[$ofs2+0x0010], c0[$ofs2+0x0088], $r1
...
```

Meta-programming!

- Big round of applause to the creator of PyCUDA: **Andreas Kloeckner** (Brown)

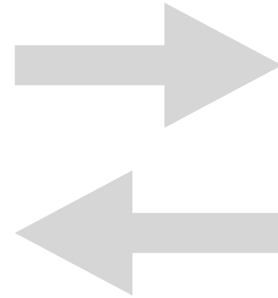


Need for speed
Hardware
Software
Science

The Approach:
Forward Engineering the Brain

REVERSE

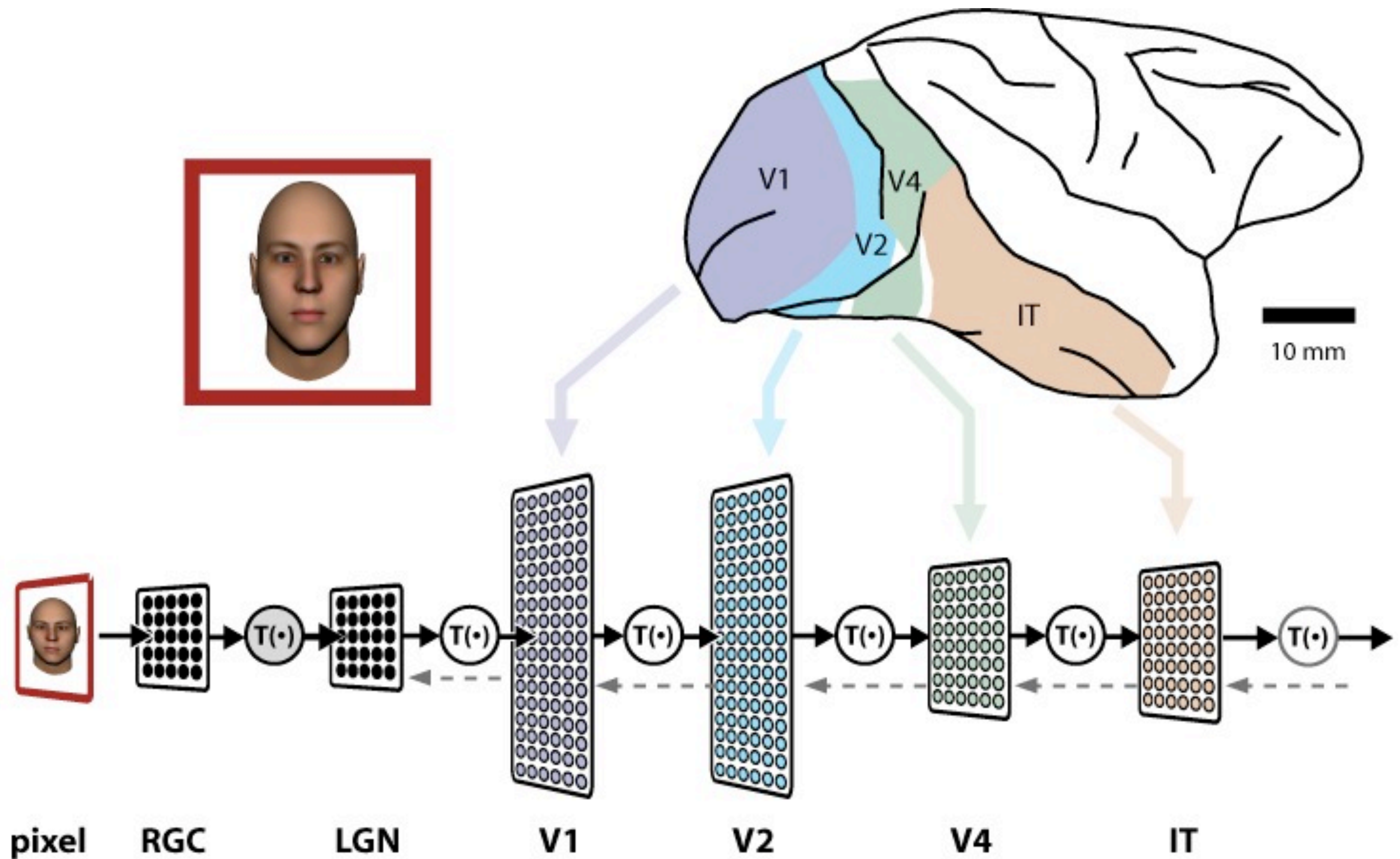
Study
Natural System



FORWARD

Build
Artificial System

Visual System



How are things done normally?

Usual Formula:

1) **One grad student**

2) **One Model** (size limited by runtime)

3) **Performance numbers on a few standard test sets**

4) **yay. we. rock.**

5) **One Ph.D.**

Doing things a little bit differently

1) One grad student

2) ~~One~~ Hundreds of Thousands of BIG Models

3) Performance numbers on a few standard test sets

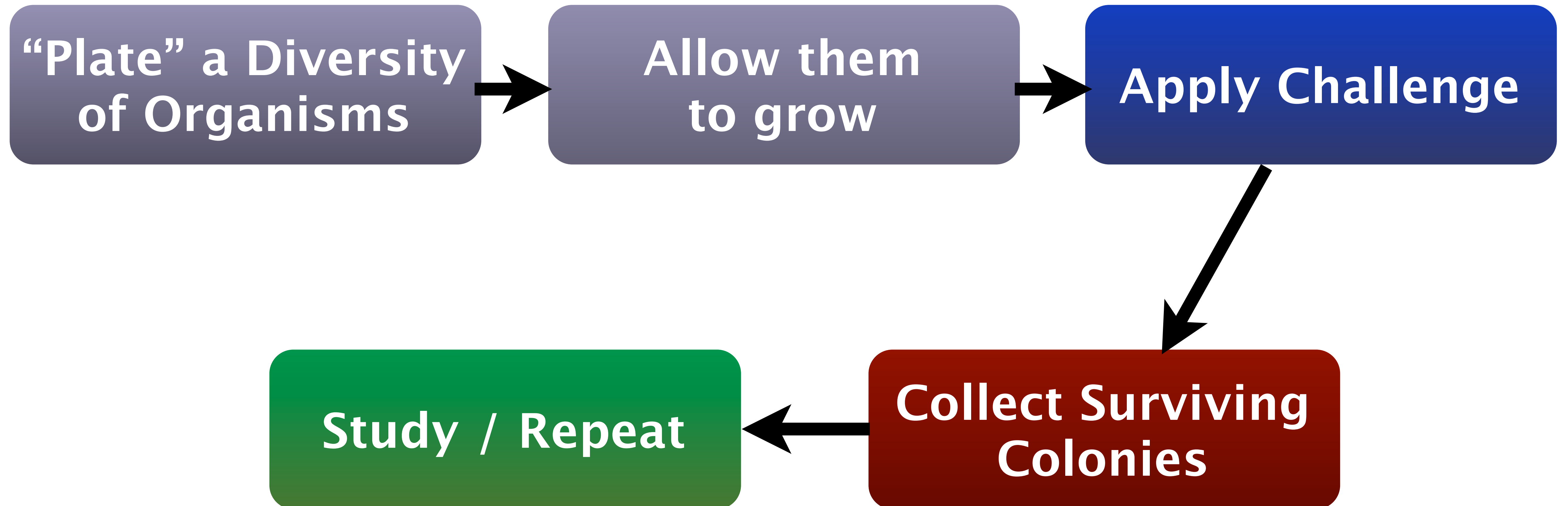
4) yay. we. rock.

5) ~~Hundreds of Thousands~~ One PhD ?

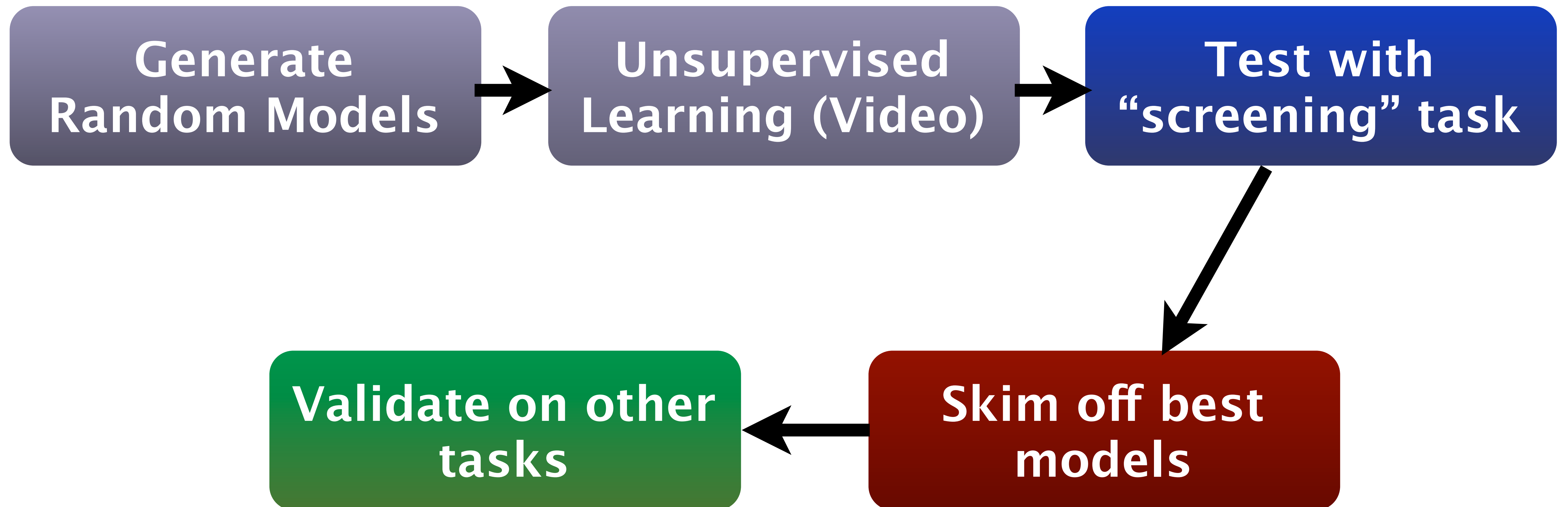
High-Throughput Screening

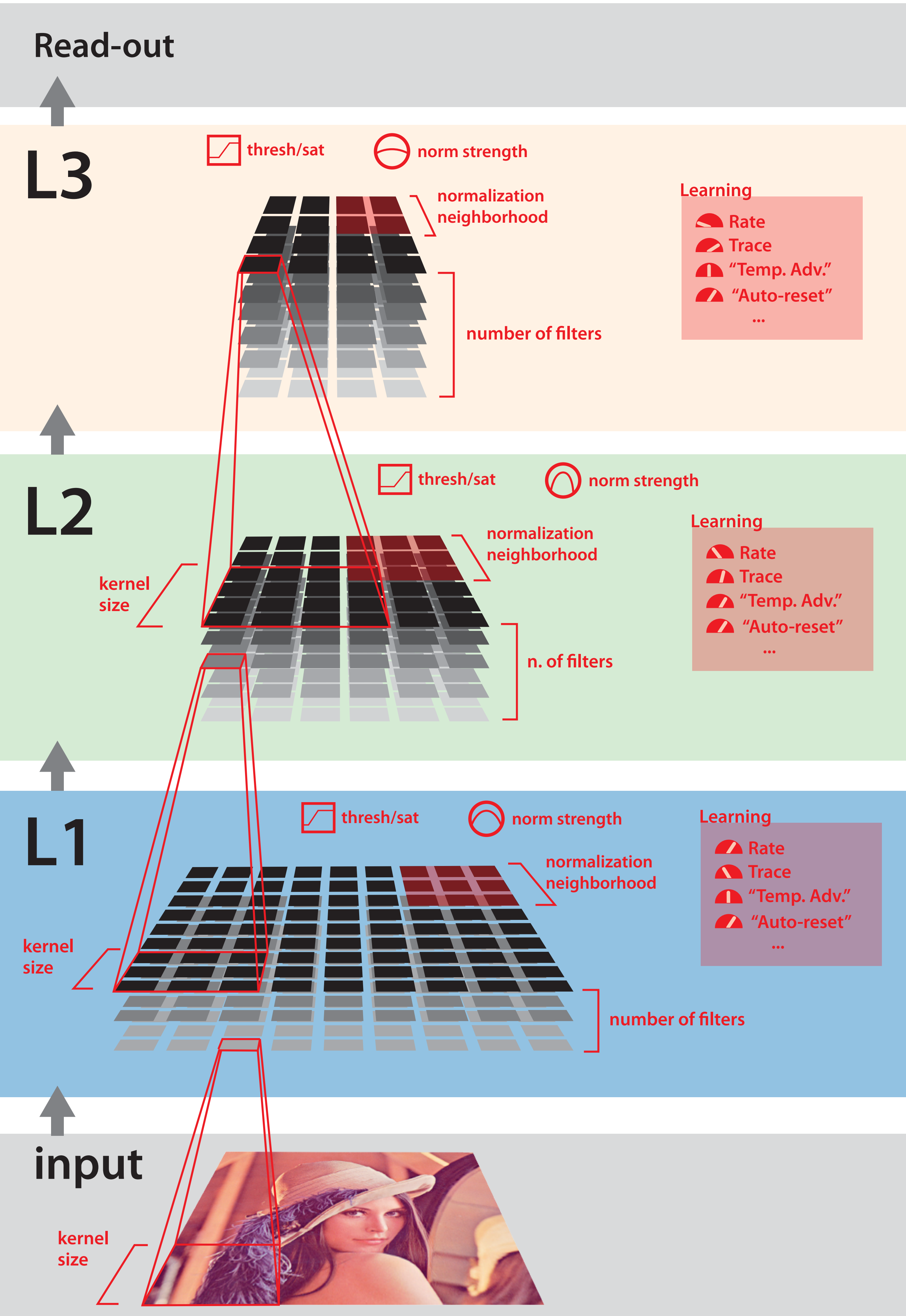
High-Throughput Screening

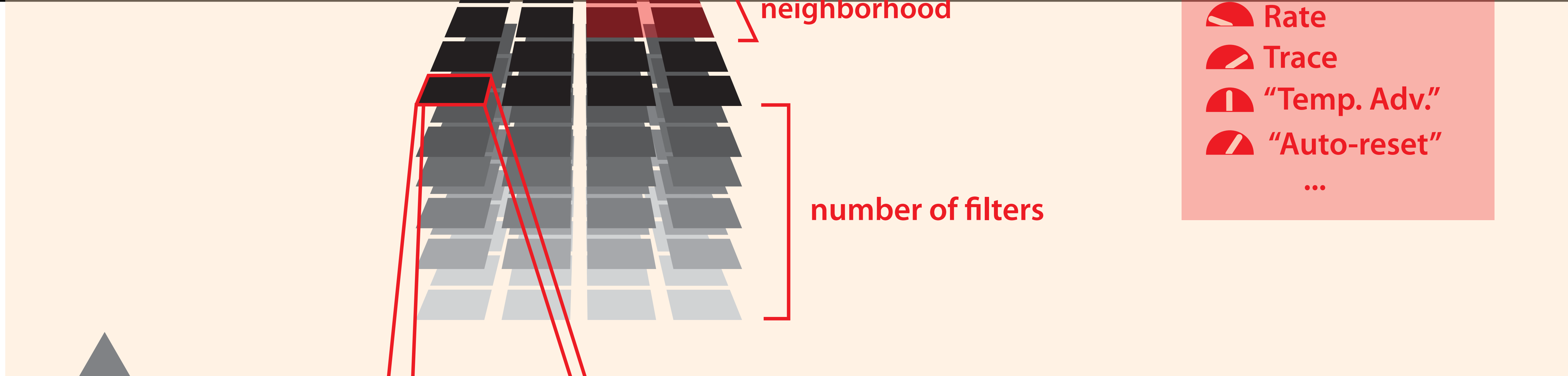
Pipeline: Biology

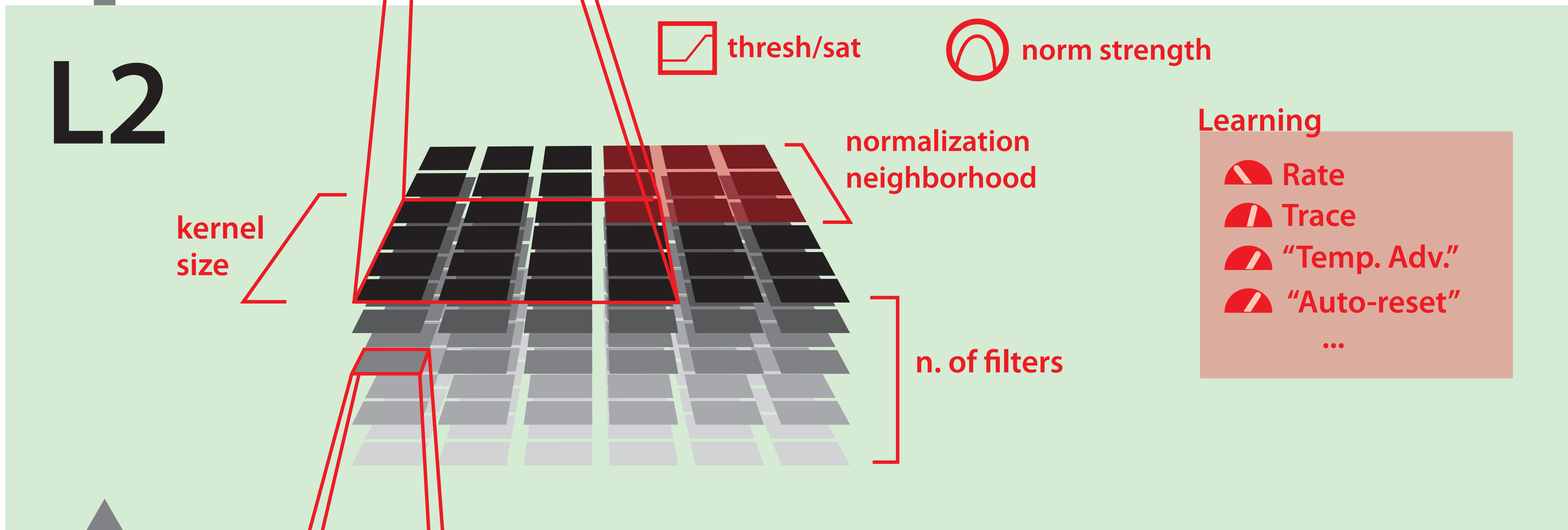


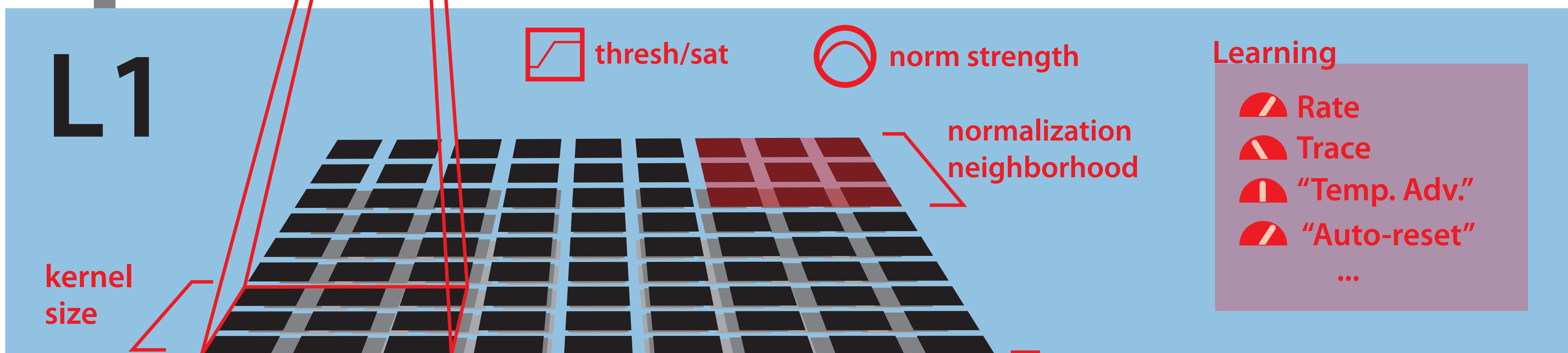
Pipeline: Biology-Inspired Vision




Read-out
L3
thresh/sat
norm strength
normalization
neighborhood
number of filters
Learning
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
L2
thresh/sat
norm strength
normalization
neighborhood
kernel
size
n. of filters
Learning
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
L1
thresh/sat
norm strength
normalization
neighborhood
kernel
size
number of filters
Learning
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
input
kernel
size


neighborhood
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
number of filters


L2
thresh/sat
norm strength
Learning
normalization
neighborhood
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
kernel
size
n. of filters


L1
thresh/sat
norm strength
Learning
normalization
neighborhood
Rate
Trace
"Temp. Adv."
"Auto-reset"
...
kernel
size

A Broad **Parametric** Model

Normalize

$N_i = Input_i / norm(Input_{neighborhd})$

Compute Filter Responses

$R_i = F_i \otimes N$

$R_i < thresh: R_i = thresh$

$R_i > sat: R_i = sat$

Determine a "Winning Filter"

$R_i' = (\sum T_k * H_k) * R_i$

winner: $\max(R_i')$

Update Filter

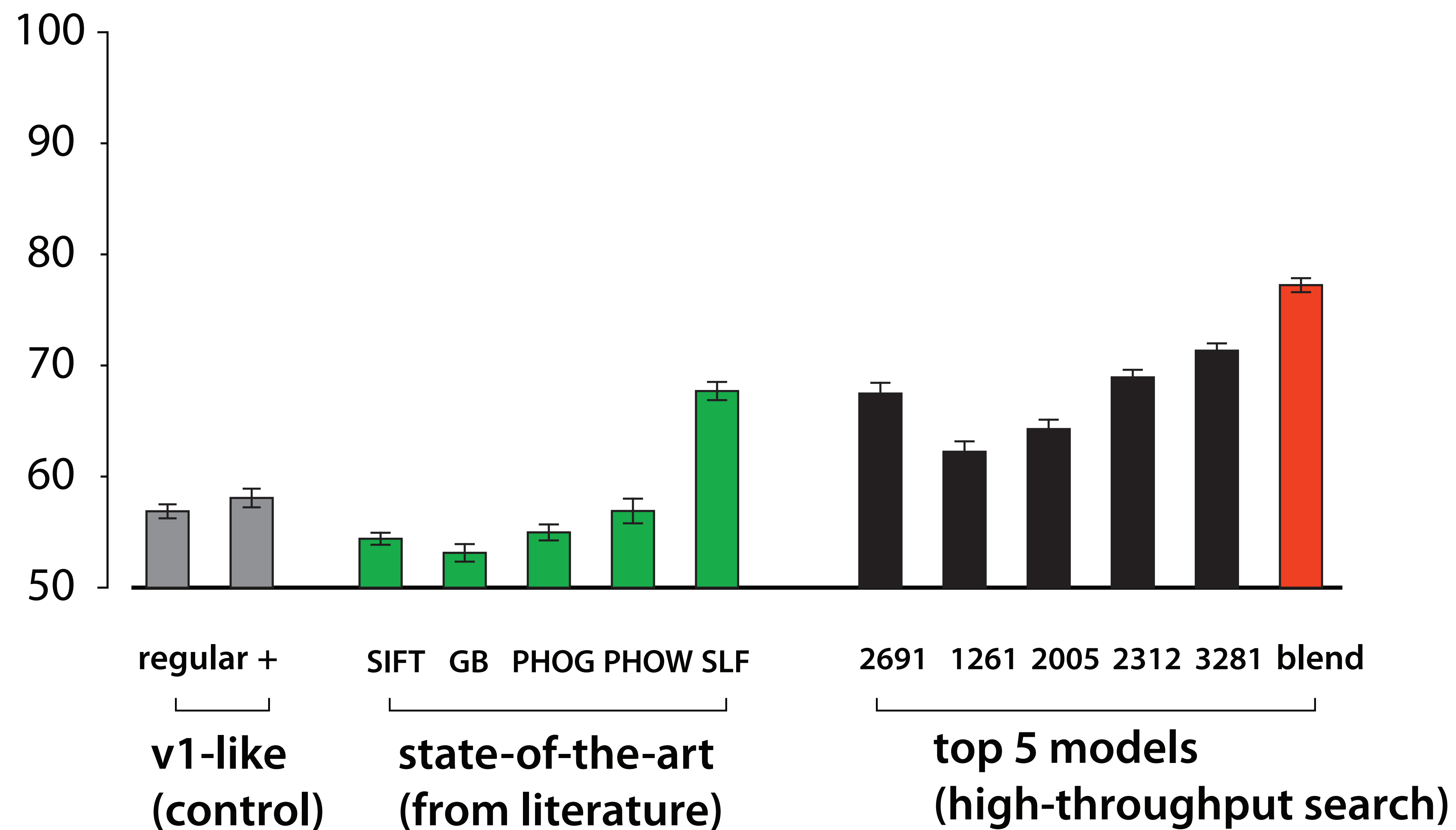
$F_{winning} = F_{winning} + \text{learning rate} * N$

- **Optimize "Coverage"**
(filters span the range of observed inputs)

- **Privilege movement of filters in certain directions using temporal information**

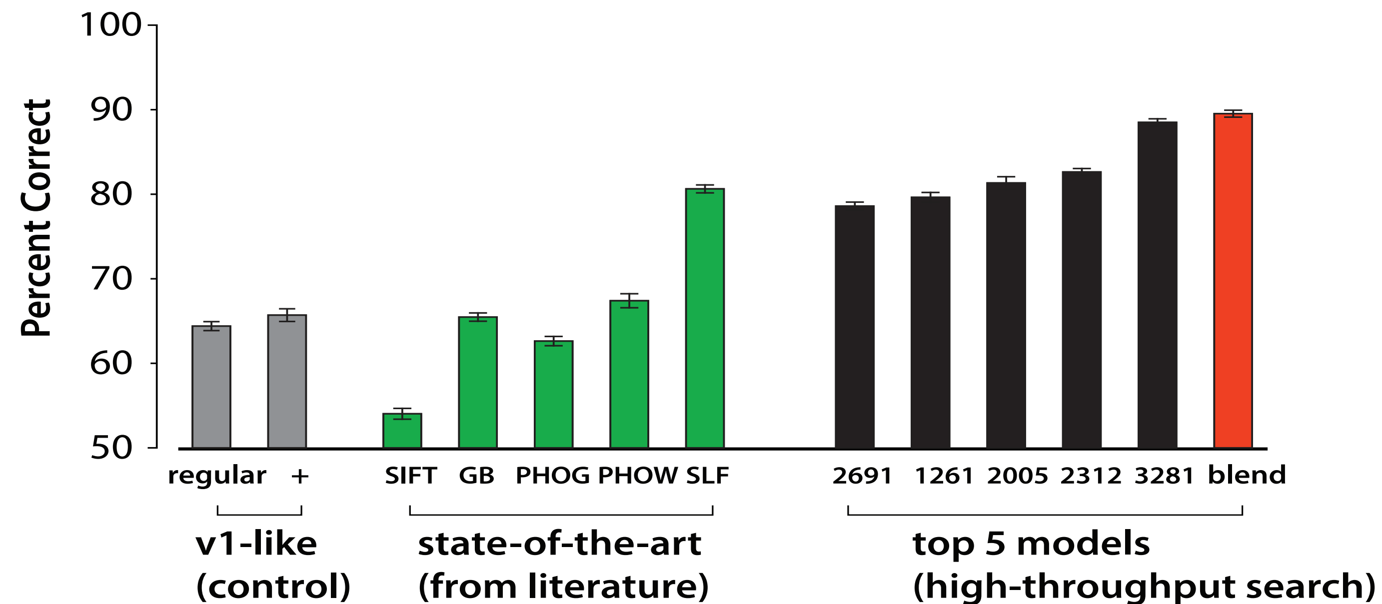
- **Expand dimensionality greatly and then scale back as layers progress**

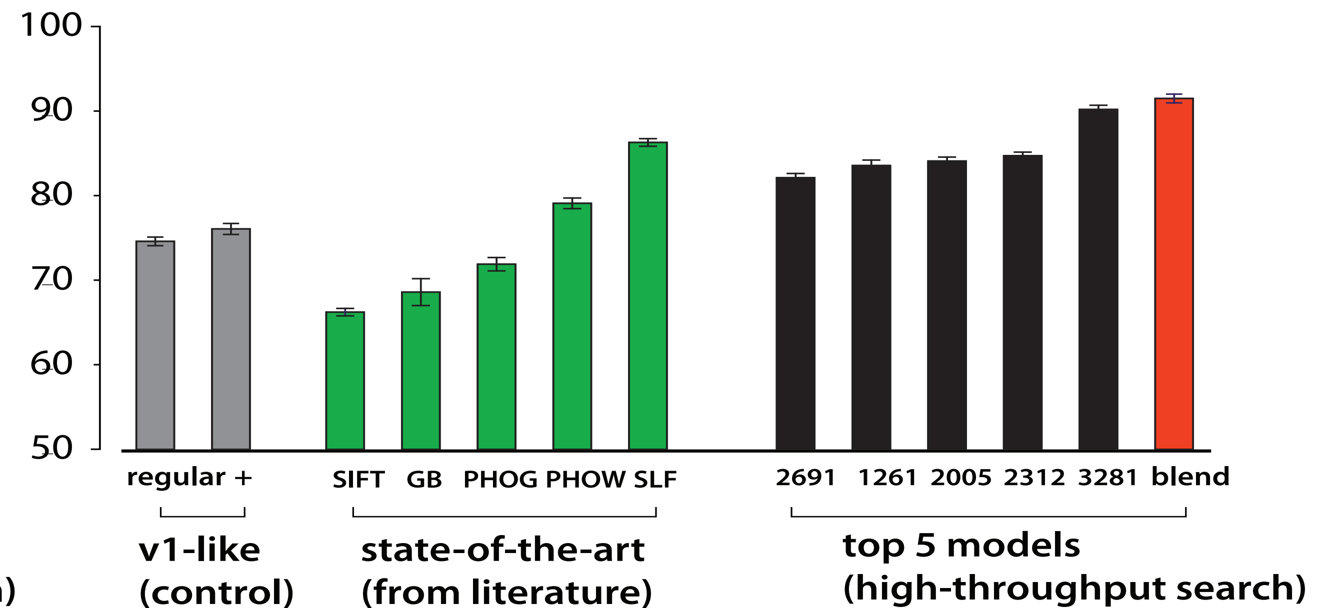
State-of-the-art performance

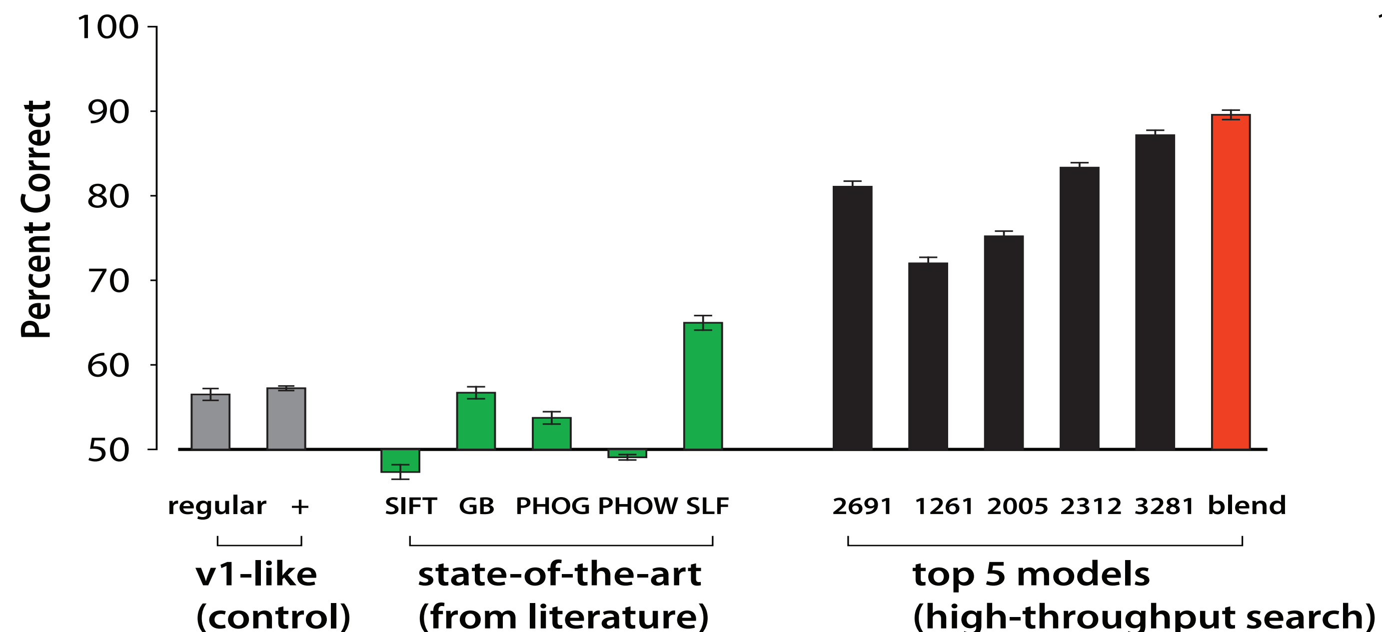


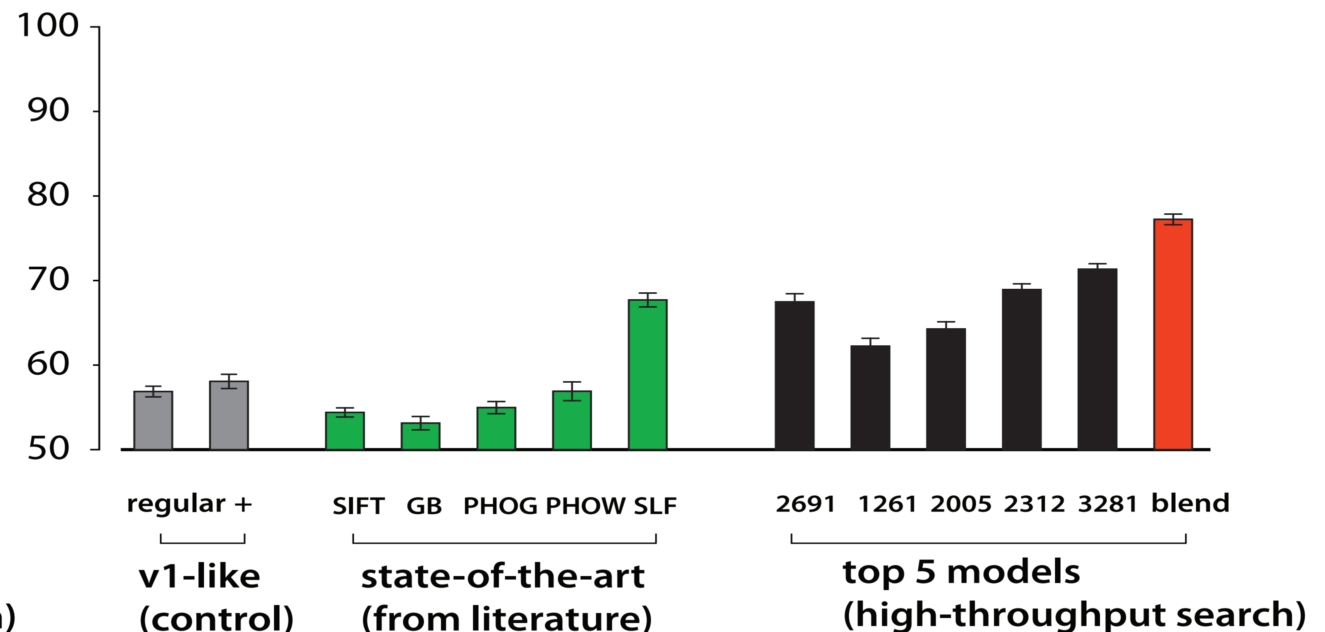
Pinto, Doukhan, DiCarlo, Cox (PLoS, in press)

State-of-the-art performance



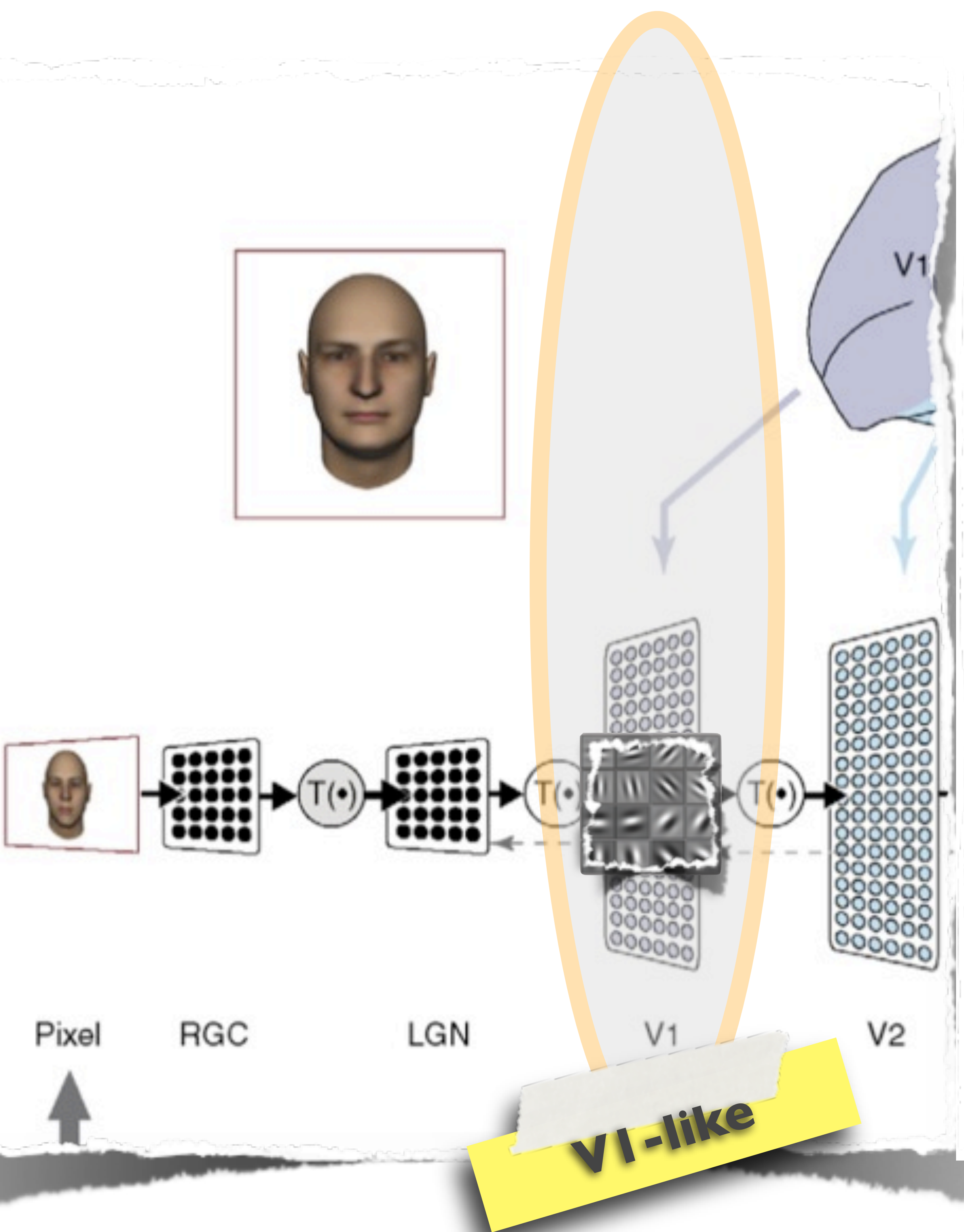






Pinto, Doukhan, DiCarlo, Cox (PLoS, in press)

State-of-the-art performance



Caltech101 Results (102 classes)

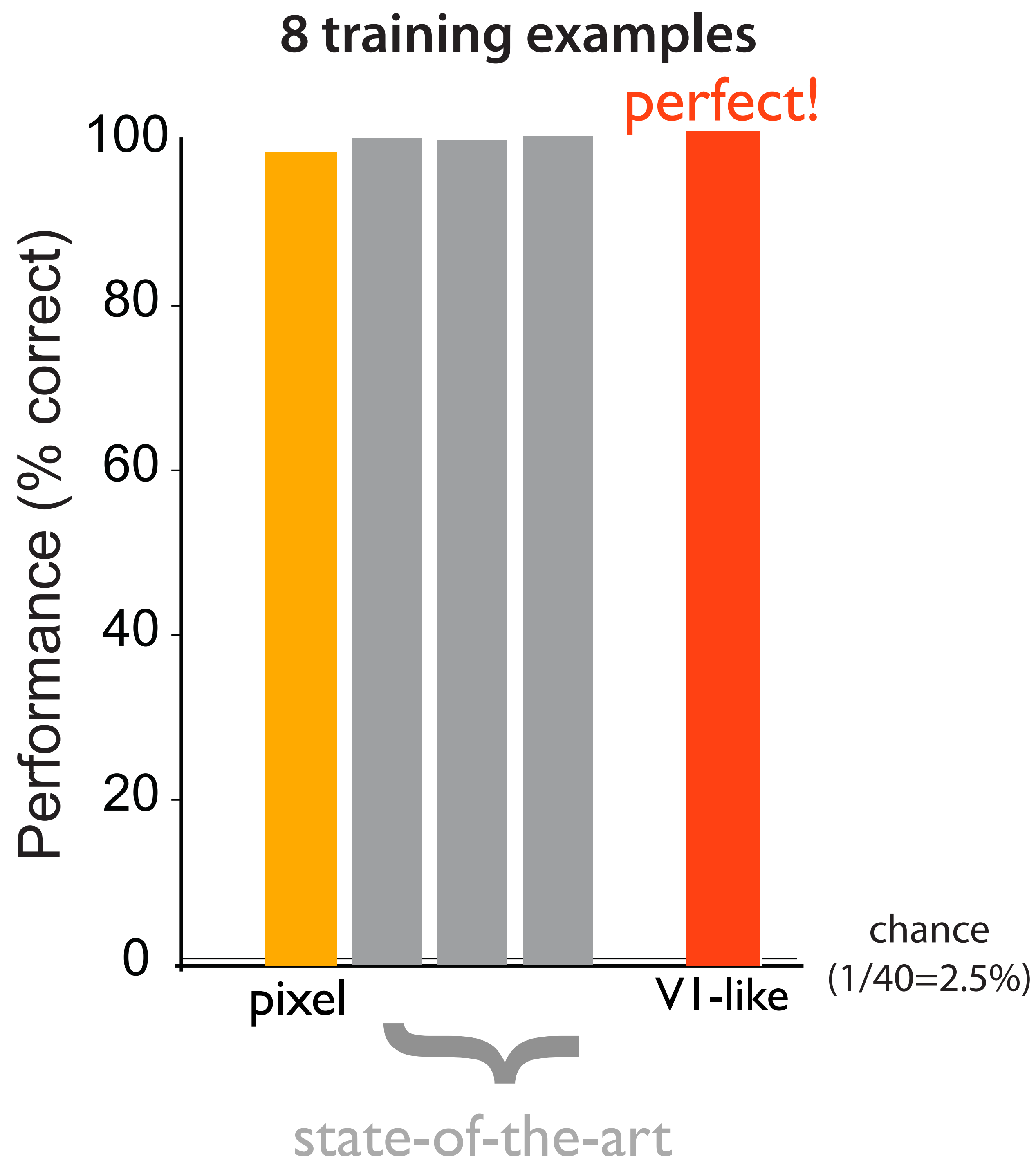
Classification performance (%)

Number of training examples

- Zhang, Berg, Maire and Malik (CVPR06)
- Lazebnik, Schmid and Ponce (CVPR06)
- Wang, Zhang and Fei-Fei (CVPR06)
- Grauman and Darrell (ICCV05)
- Mutch and Lowe (CVPR06)
- V1S+

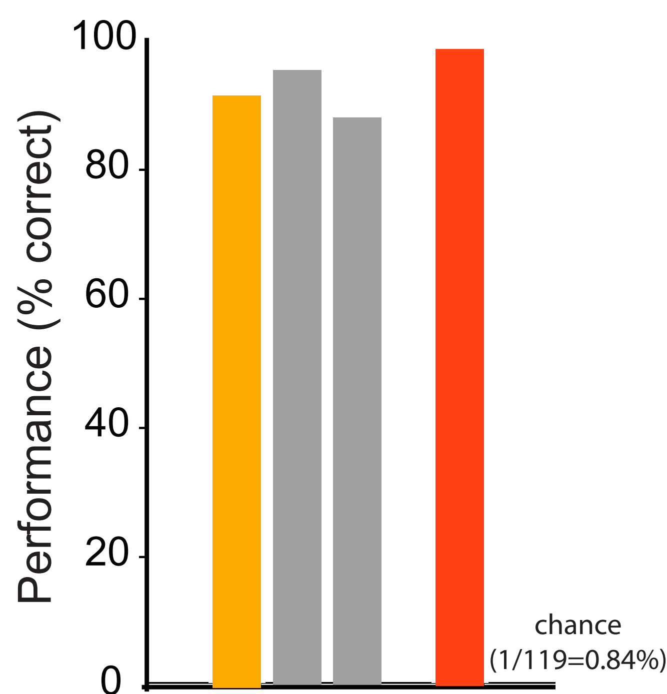
Pinto, Cox, DiCarlo PLoS08

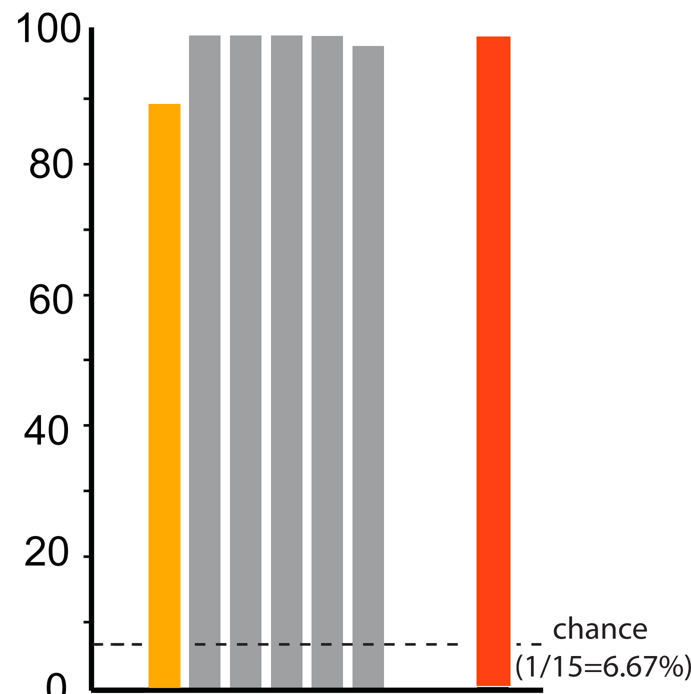
State-of-the-art performance

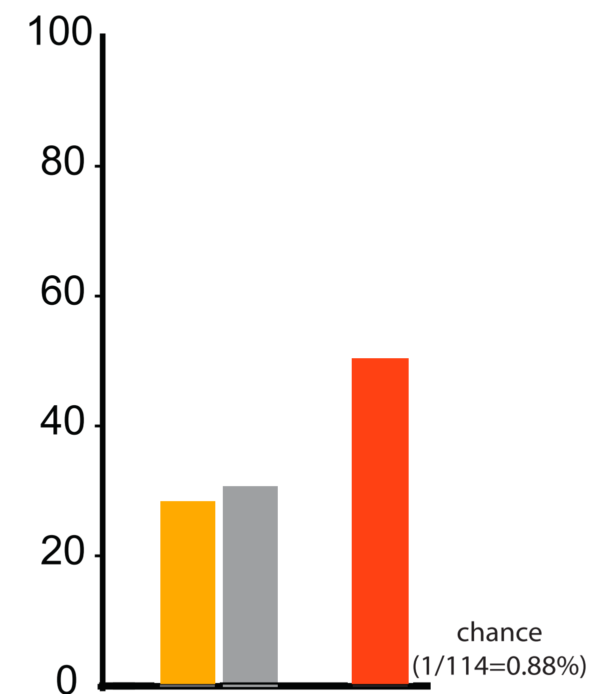


Pinto, DiCarlo, Cox ECCV08

State-of-the-art performance







state-of-the-art V1-like pixels

Pinto, DiCarlo, Cox ECCV08

State-of-the-art performance

LFW Face Set

Reference	Methods	Performance
Huang08 [6]	Nowak [8]	73.93%±0.49
	MERL	70.52%±0.60
	Nowak+MERL	76.18%±0.58
Wolf08 [17]	descriptor-based	70.62%±0.57
	one-shot-learning*	76.53%±0.54
	hybrid*	78.47%±0.51
This paper	Pixels/MKL	68.22%±0.41
	V1-like/MKL	**79.35%±0.55**

Table 3. Average performance comparison with the current state-of-the-art on LFW. *note that the "one-shot-learning" and "hybrid" methods from [17] can't directly be compared to ours as they exploit the fact that individuals in the training and testing sets are mutually exclusive (i.e. using this property, you can build a powerful one-shot-learning classifier knowing that each test example is *different* from all the training examples, see [17] for more details. Our decision not to use such techniques effectively handicaps our results relative to reports that use them).

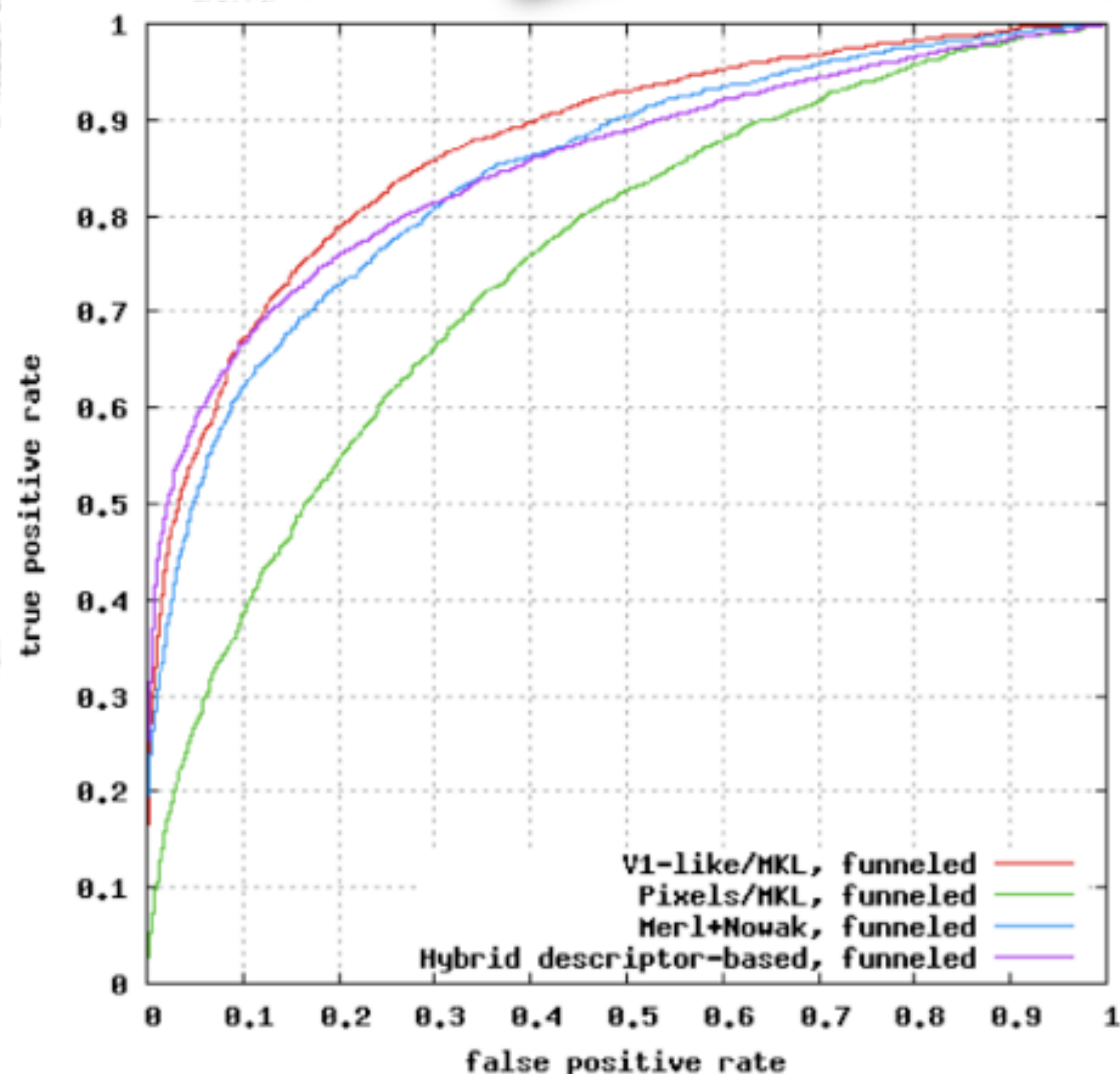


Figure 1. ROC curve comparison with the current state-of-the ar
on LFW. These curves were generated using the standard proce
dure described in [24].

Pinto, DiCarlo, Cox CVPR09

DiCarlo Lab @ MIT

The Visual Neuroscience Group
@ The Rowland Institute at Harvard

Acknowledgements

NVIDIA®

"Thank You"
COME AGAIN
and bring
a Friend!