

Fork me on GitHub

End-to-end

Deep Learning *of* **Optimization Heuristics**

<http://chriscummins.cc/pact17>



Chris Cummins

University of Edinburgh



Pavlos Petoumenos

University of Edinburgh



Zheng Wang

Lancaster University



Hugh Leather

University of Edinburgh



THE UNIVERSITY of EDINBURGH
informatics

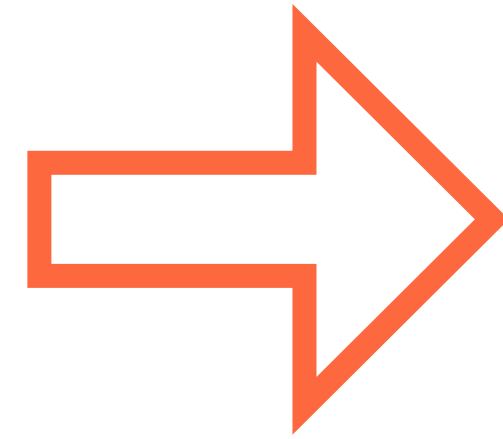
EPSRC Centre for Doctoral Training in
Pervasive Parallelism

EPSRC

Engineering and Physical Sciences
Research Council

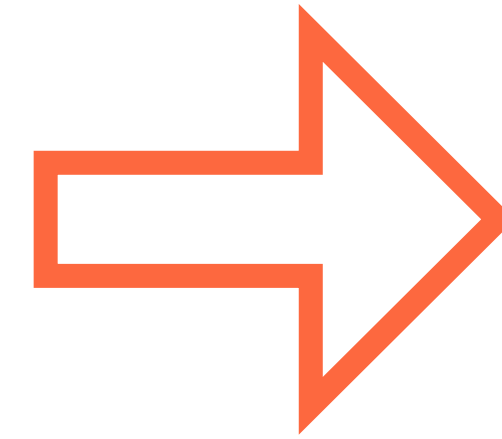
compilers are very complex

```
int main(  
  int argc,  
  char** arg)  
{...
```



hundreds,
thousands,
millions

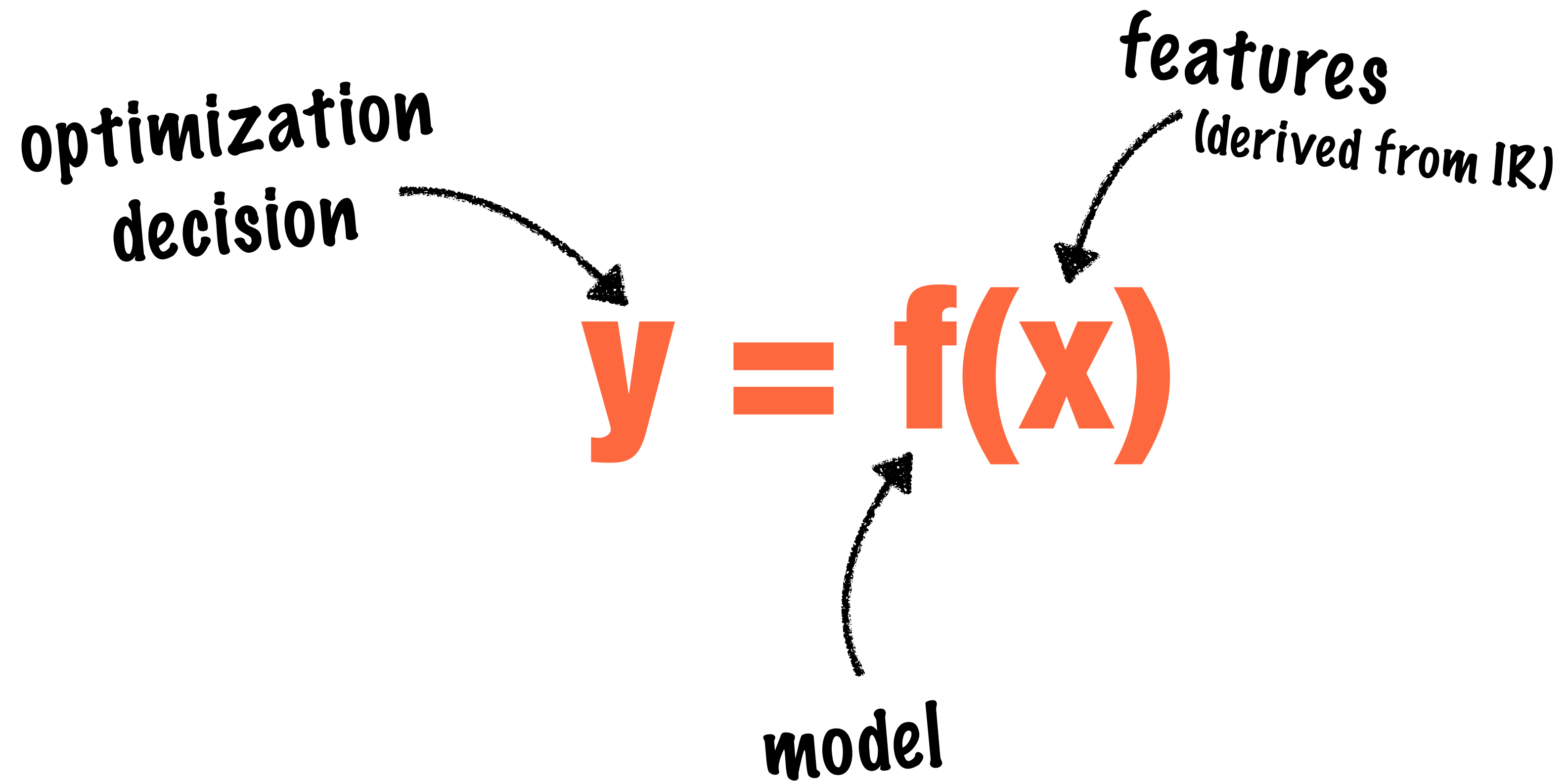
of choices



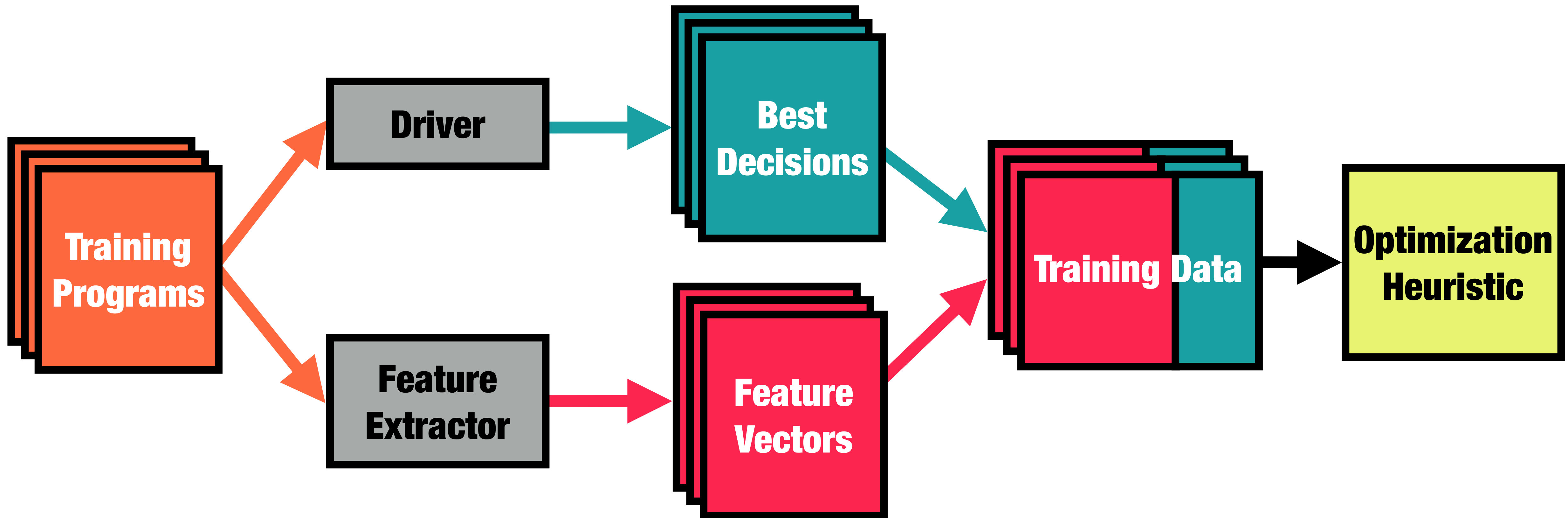
```
_main:  
  .cfi_start  
proc  
## BB#0:  
  pushq %rbp  
...
```

hand-coded heuristics
(out of date by time of release)

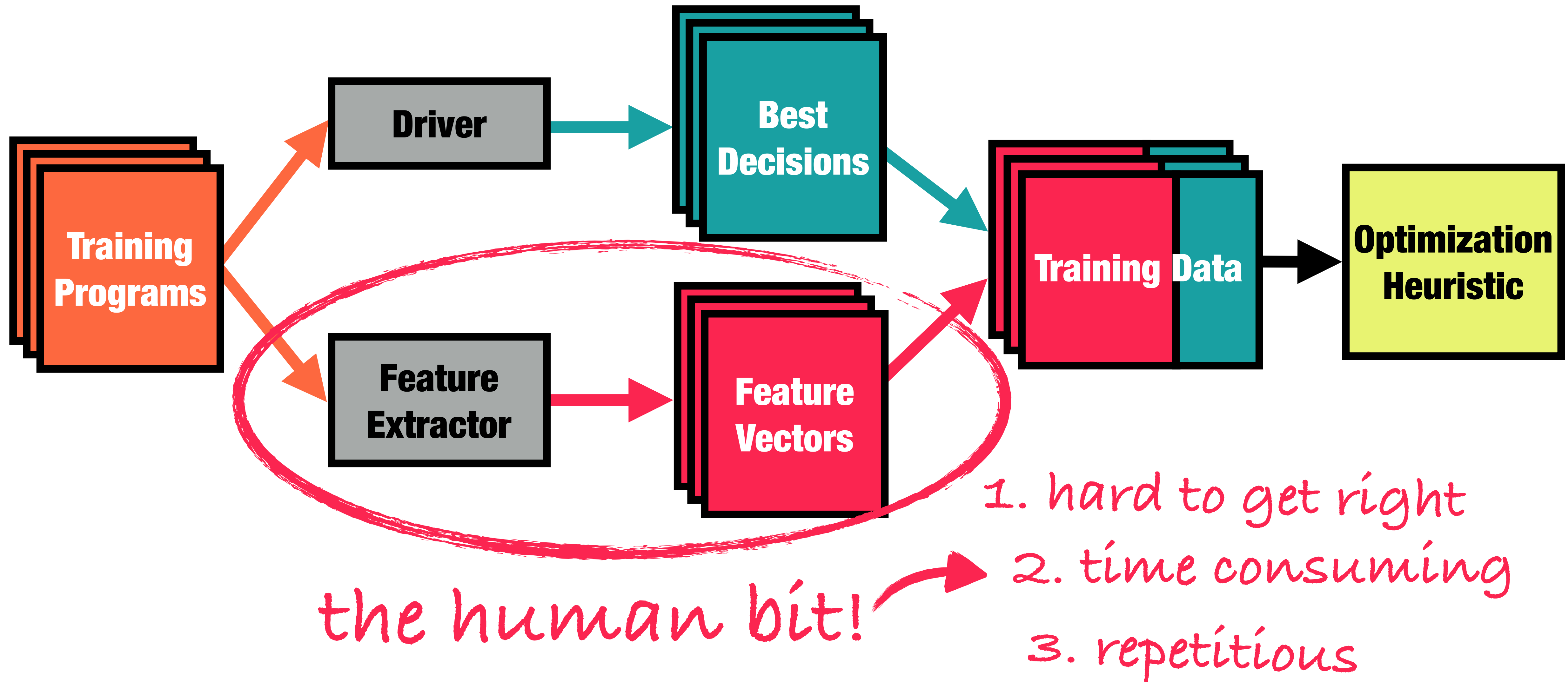
Machine learning in compilers



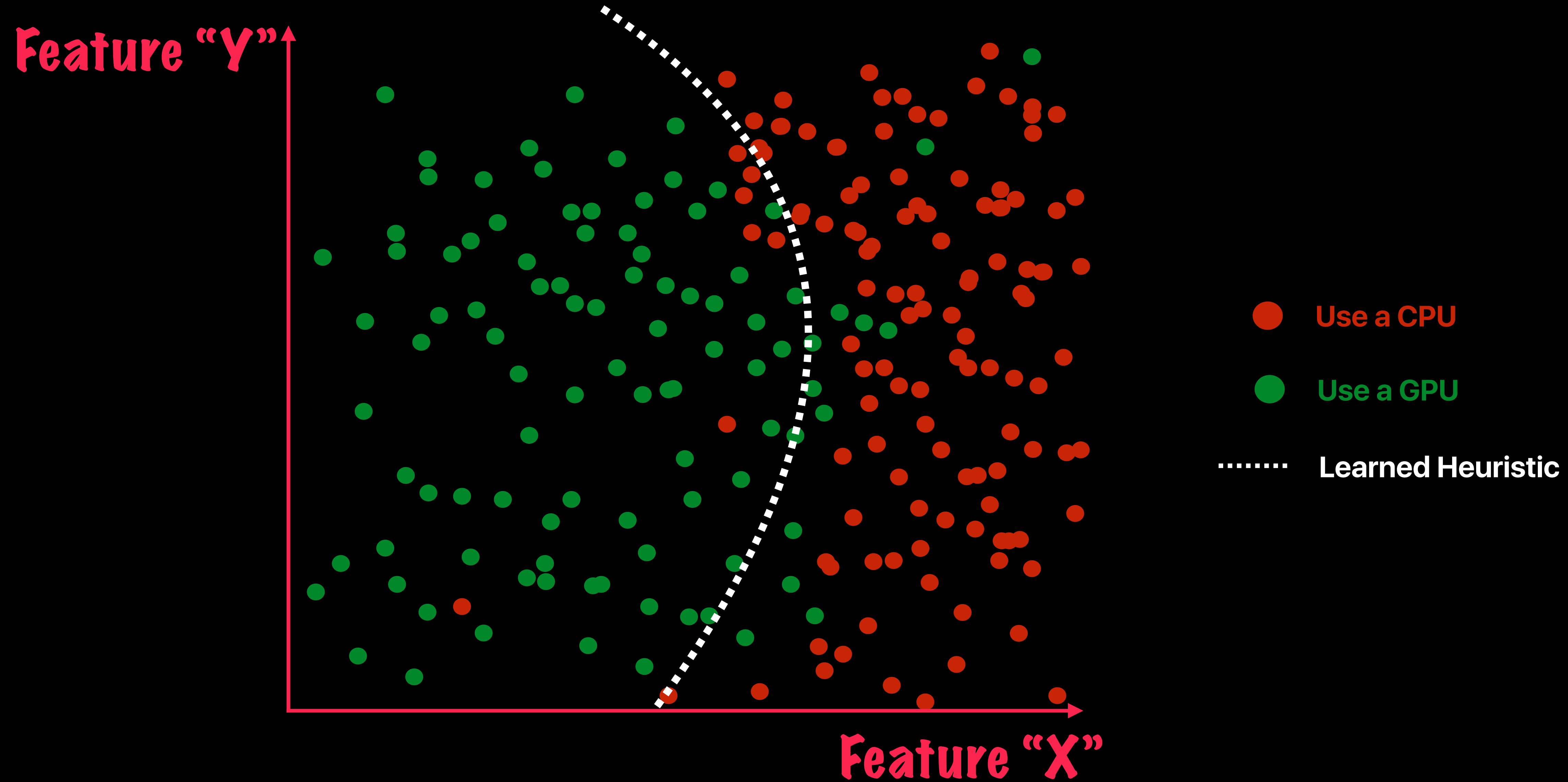
Machine learning in compilers



Machine learning in compilers



Feature space



Feature space

Feature "y"

need good
features!

● Use a CPU

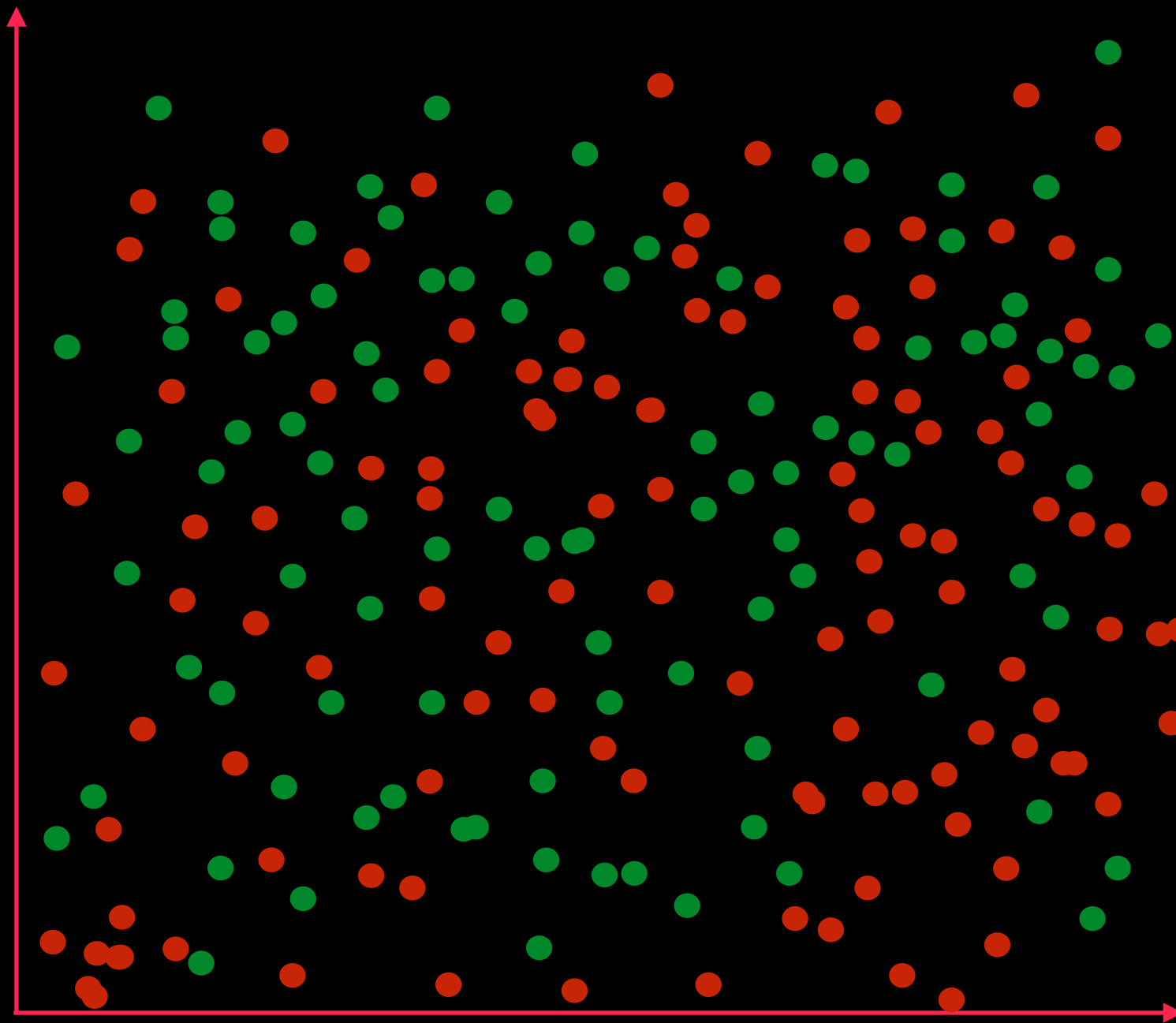
● Use a GPU

..... Learned Heuristic

Feature "x"

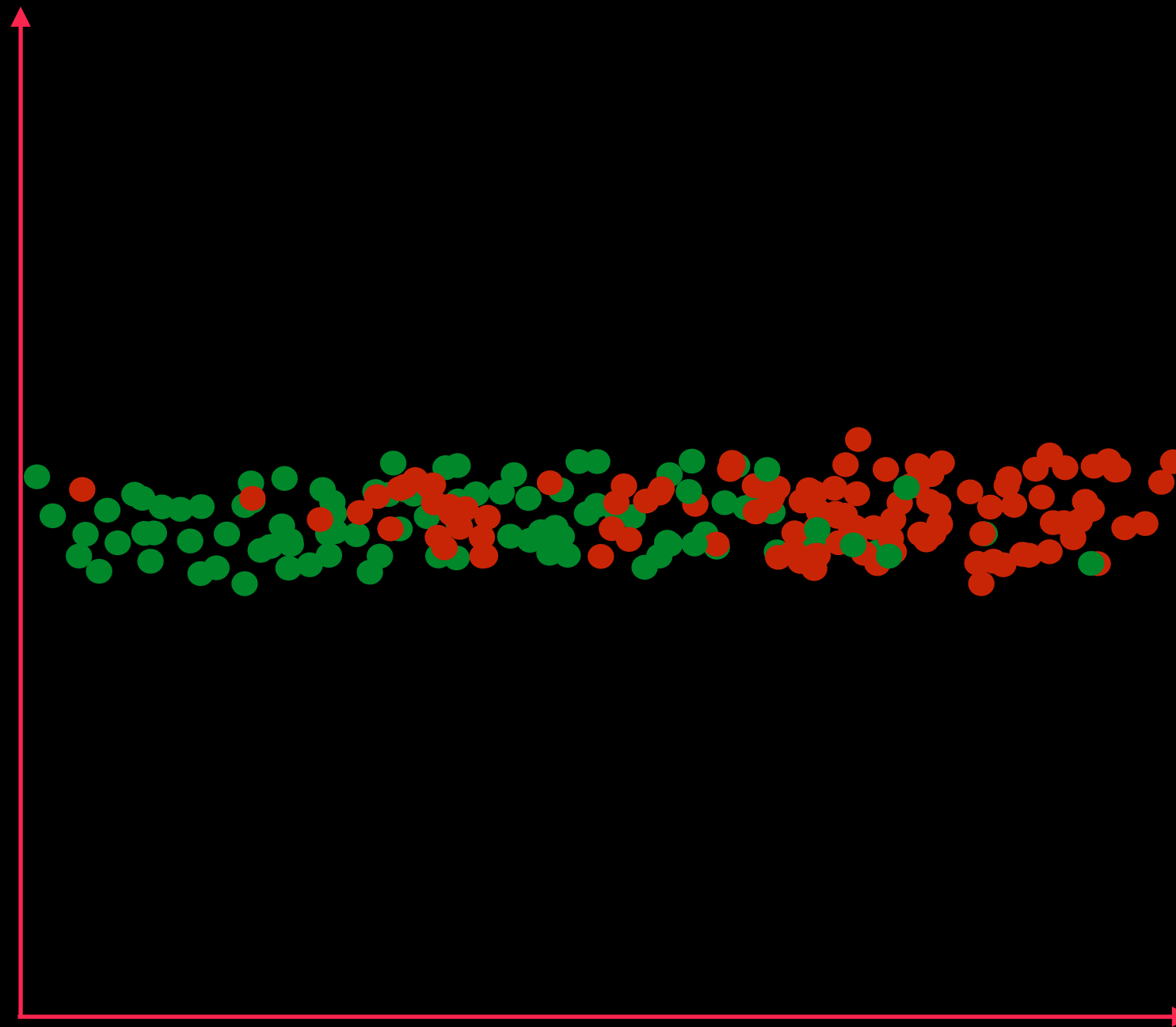


Ways to fail



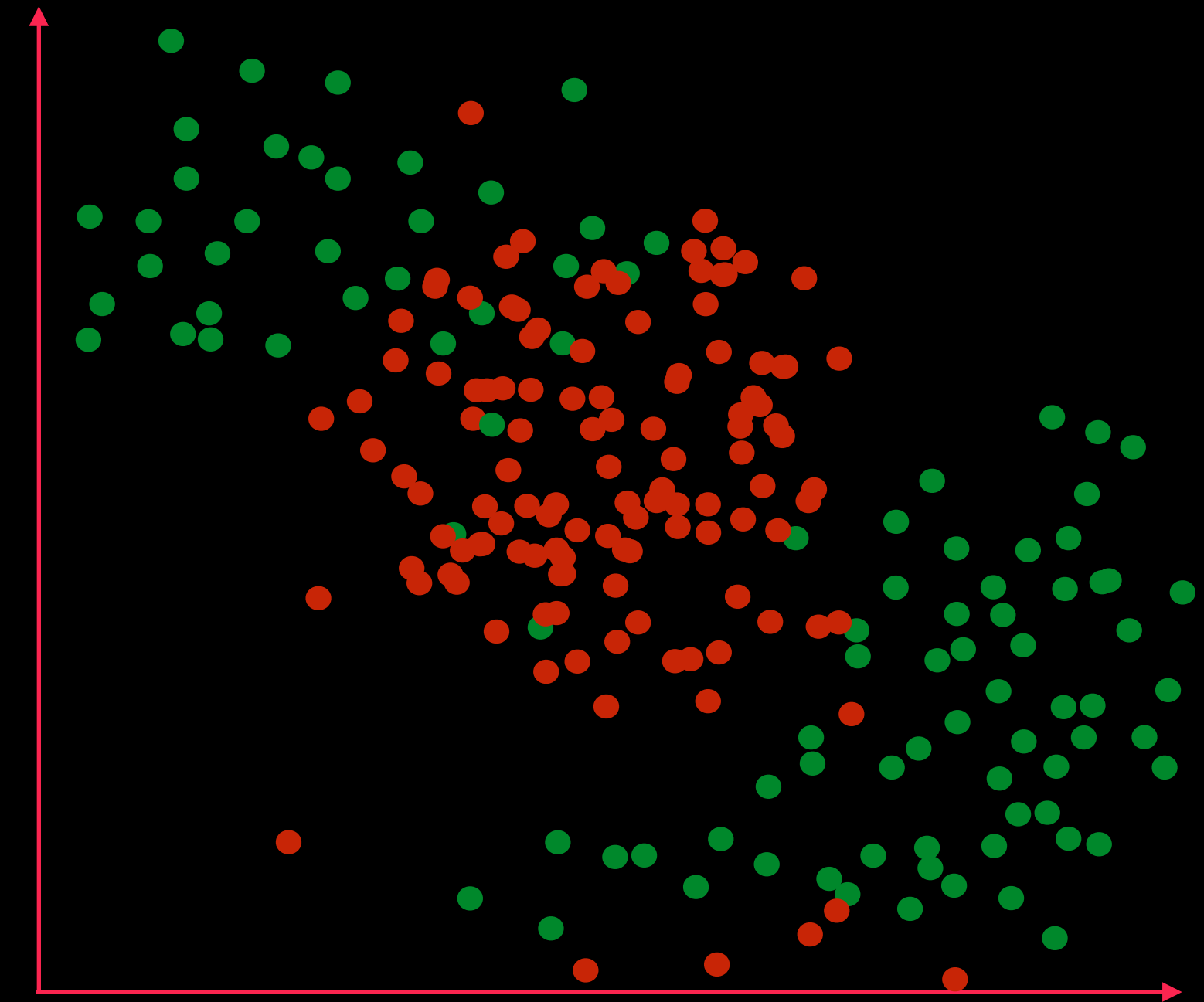
irrelevant

e.g. not capturing the right information



incomplete

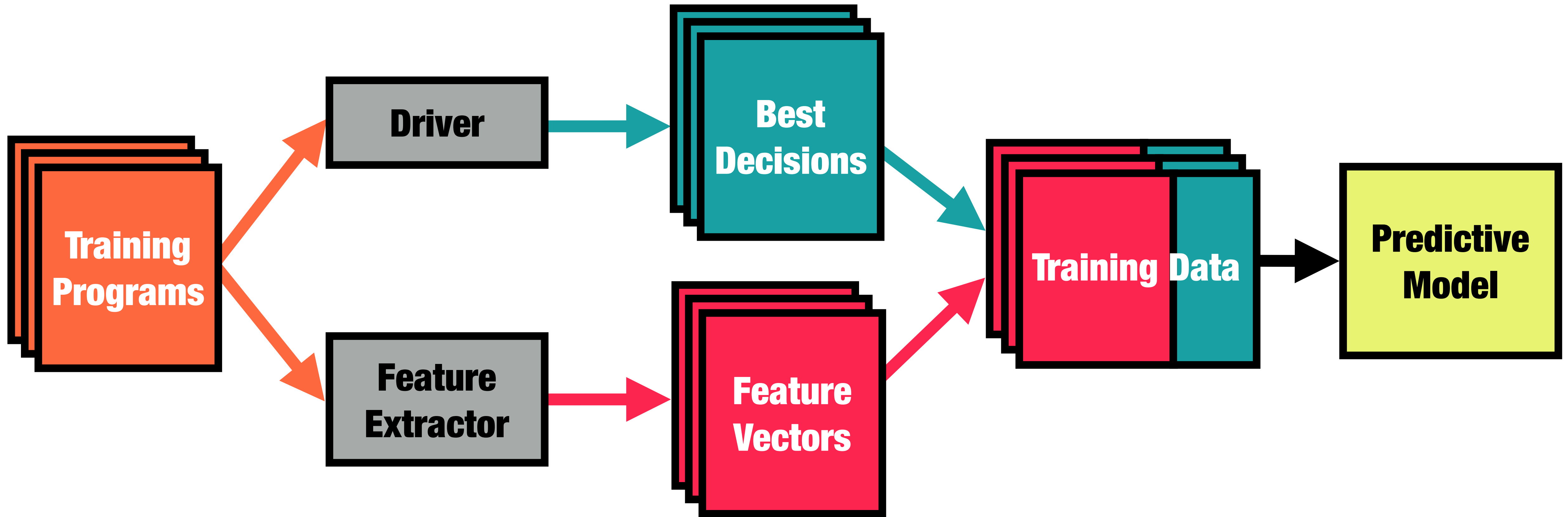
e.g. missing critical information



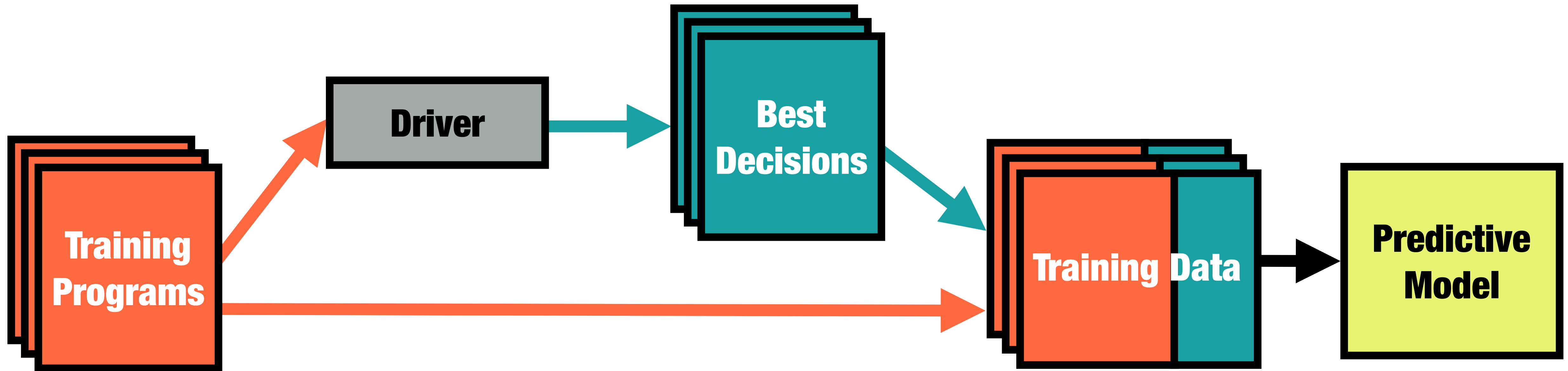
unsuitable

e.g. wrong combination of features / model

What we have



What we need



Contributions

Heuristics without features

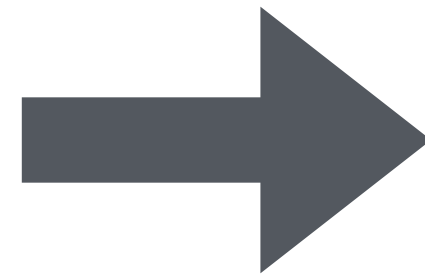
Beats expert approach

Learning across heuristics

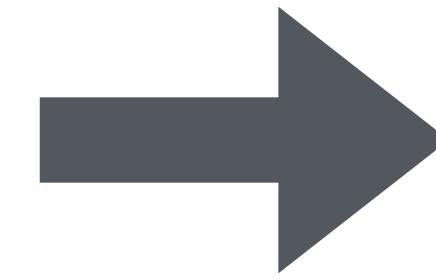
Our approach

```
int  
main(int  
argc, char  
**argv)  
{ ...
```

**Program
Code**



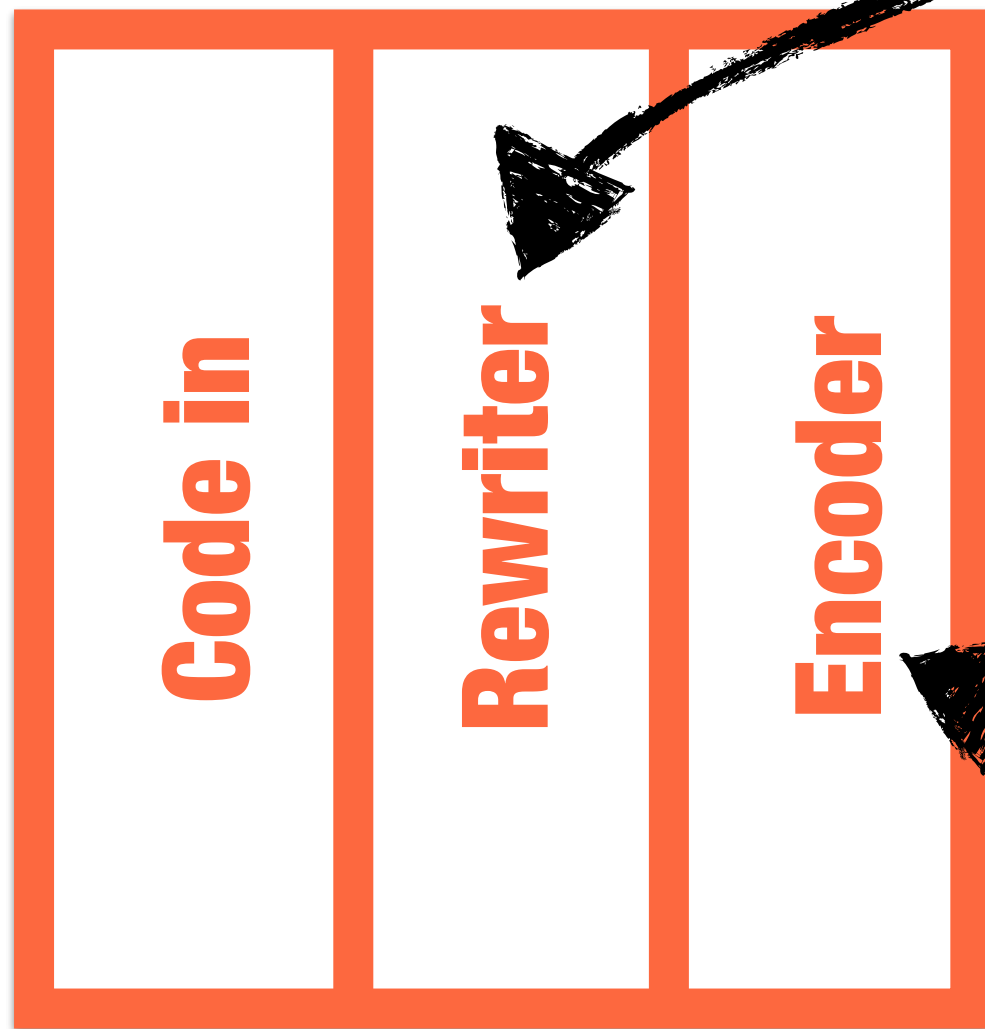
**Deep
Learning**



**Optimization
Decision**

Our approach

preprocessing



normalize identifiers & code style

- 1.var/fun names: 'foo', 'bar', ... to 'a', 'b', ...
- 2.sanitize whitespace
- 3.consistent use of optional braces

encode as sequence of vocabulary indices

Vocabulary table for characters +
lang keywords

Deep
Learning

Optimization
Decision

Program
Code

Our approach

map vocab indices
into real space

summarize sequence as vector
(2 layer LSTM network)

Embedding

Language
Model

Heuristic
Model

predict optimization
on vector (2 layer DNN)

Deep
Learning

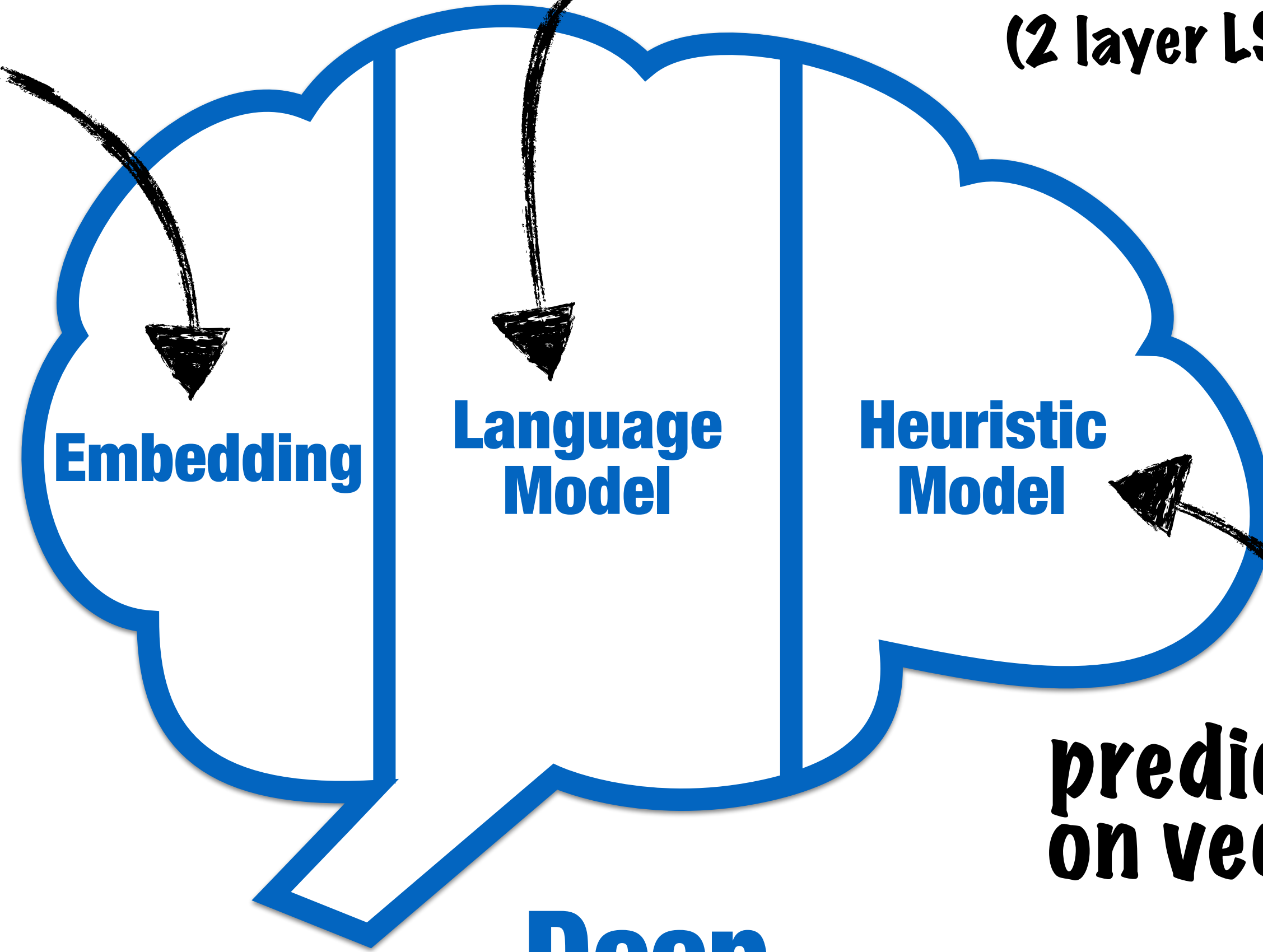
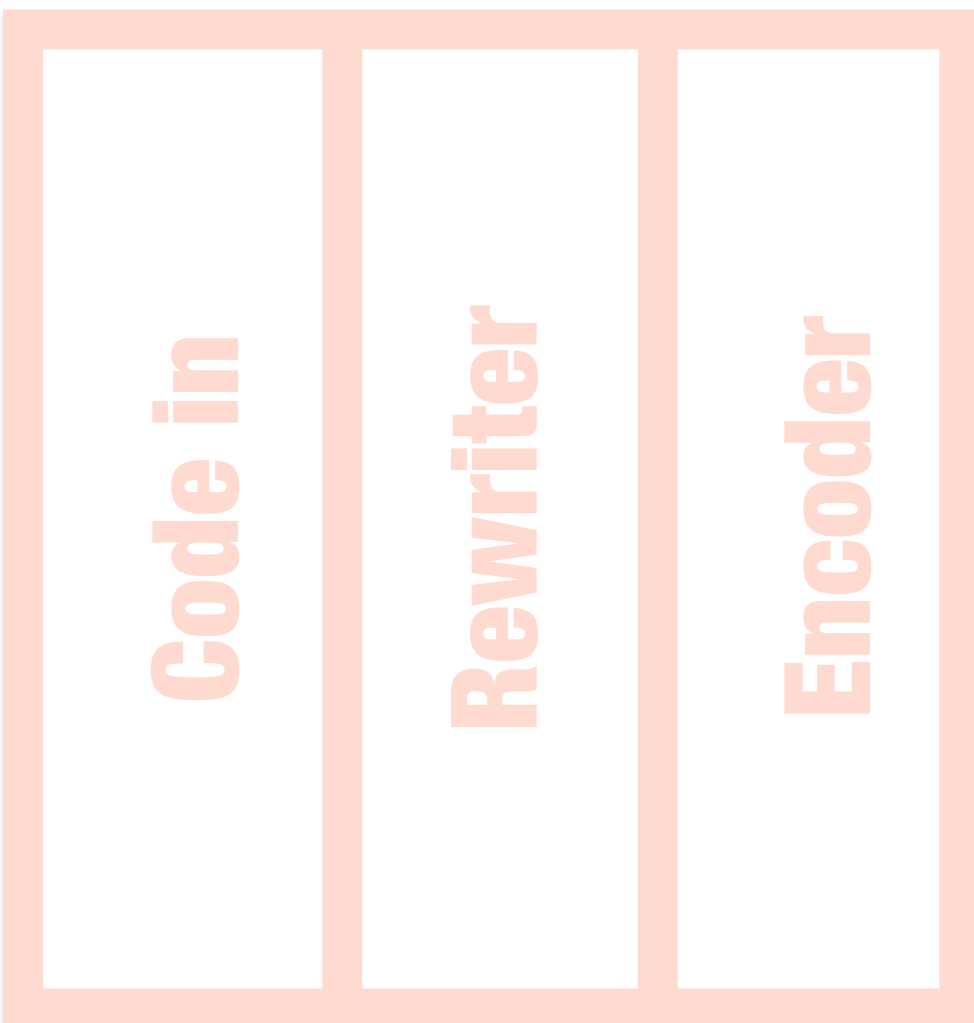
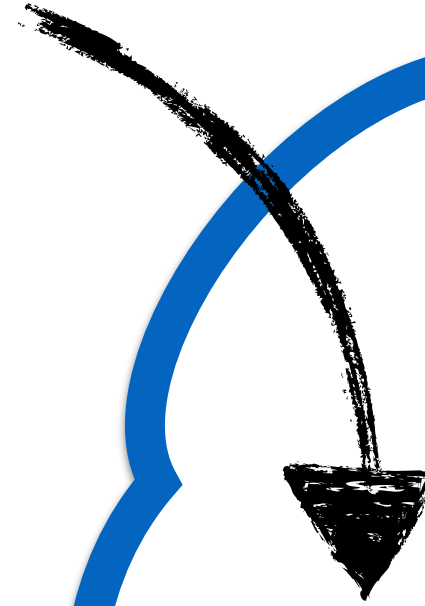
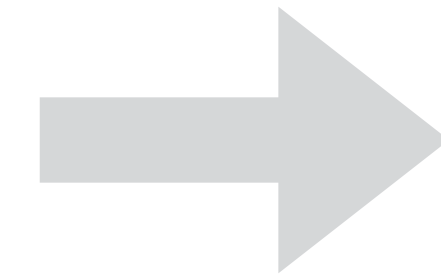
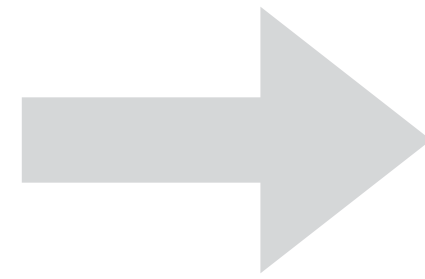
Optimization
Decision

Code in

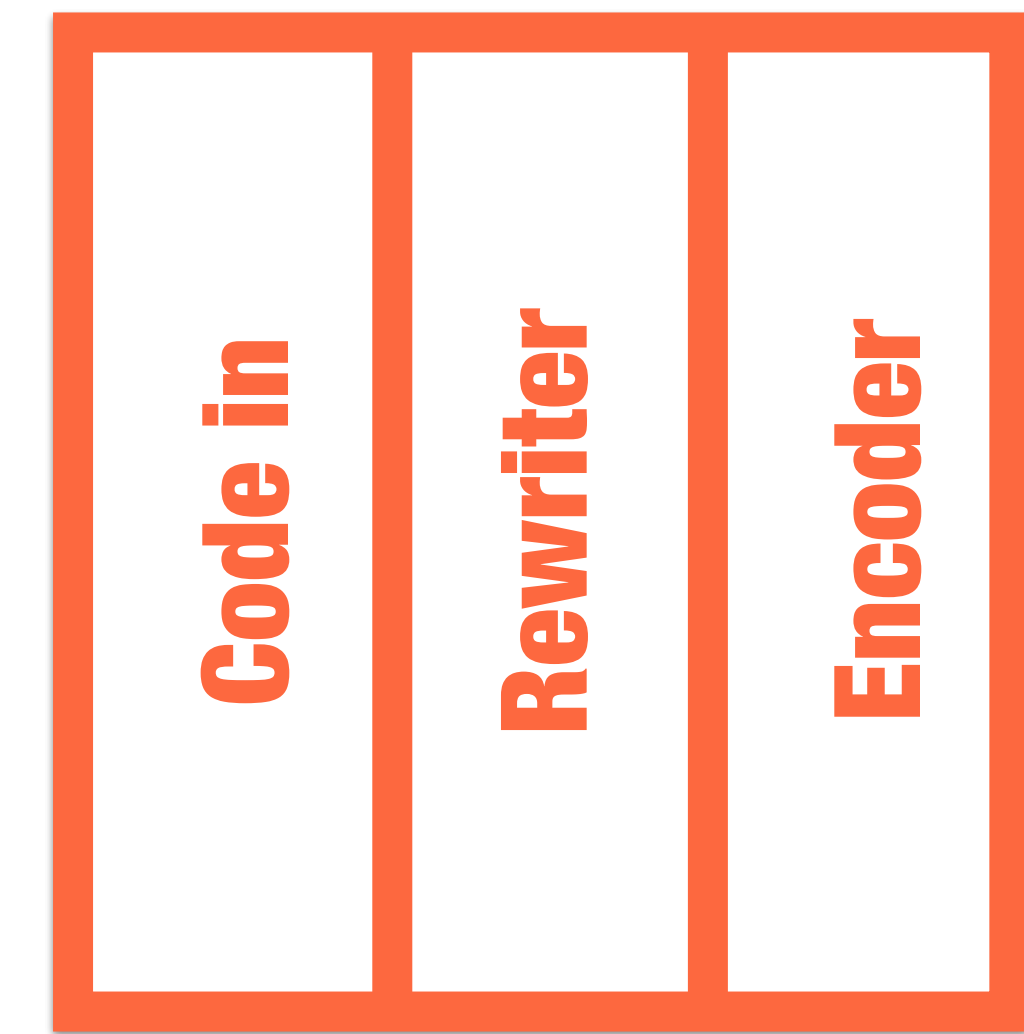
Rewriter

Encoder

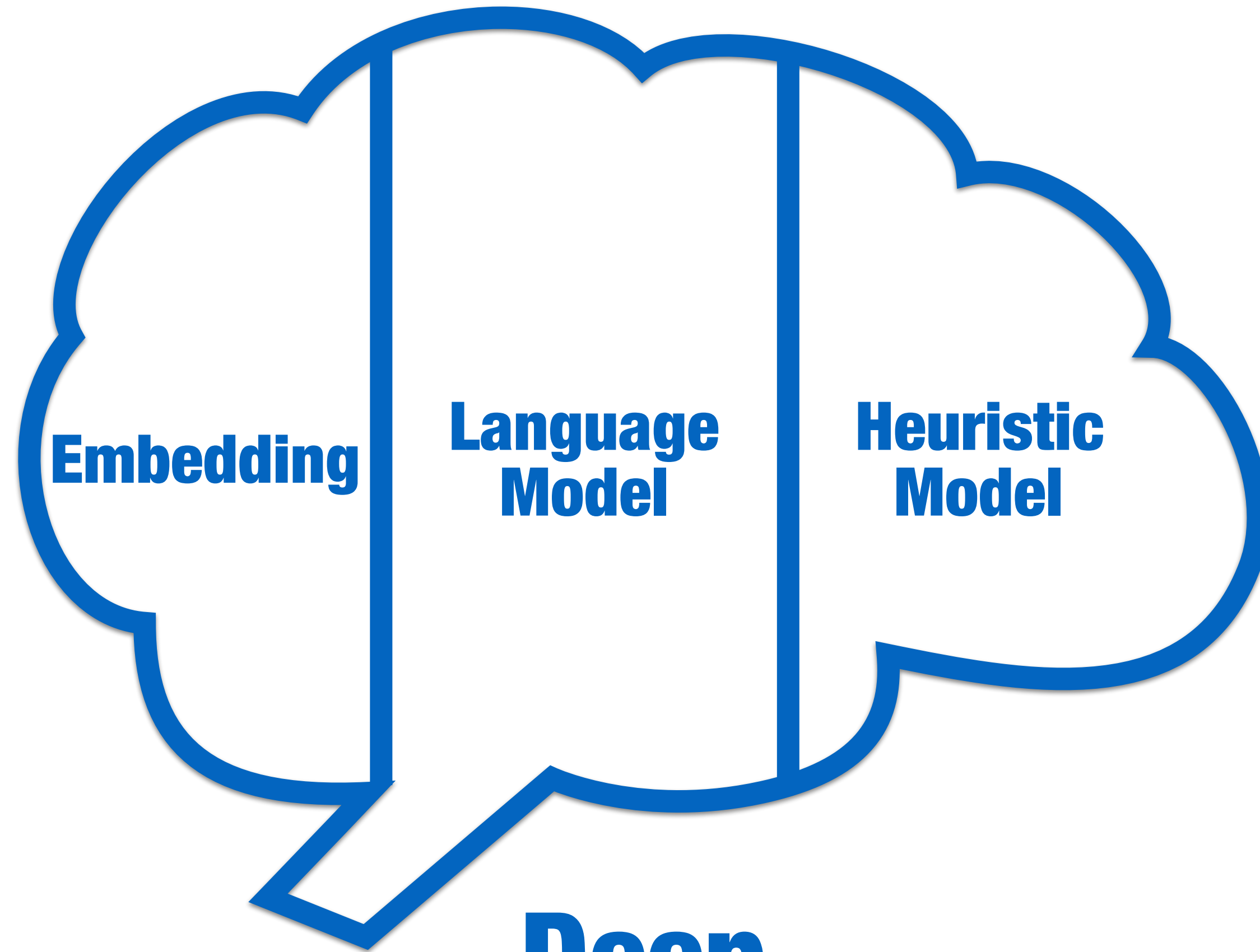
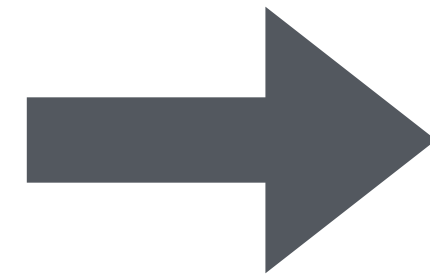
Program
Code



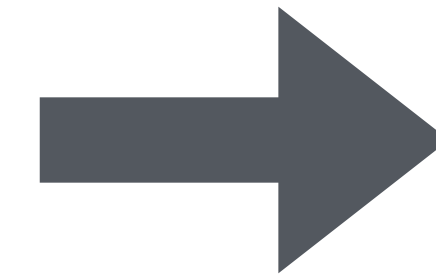
Our approach



**Program
Code**



**Deep
Learning**

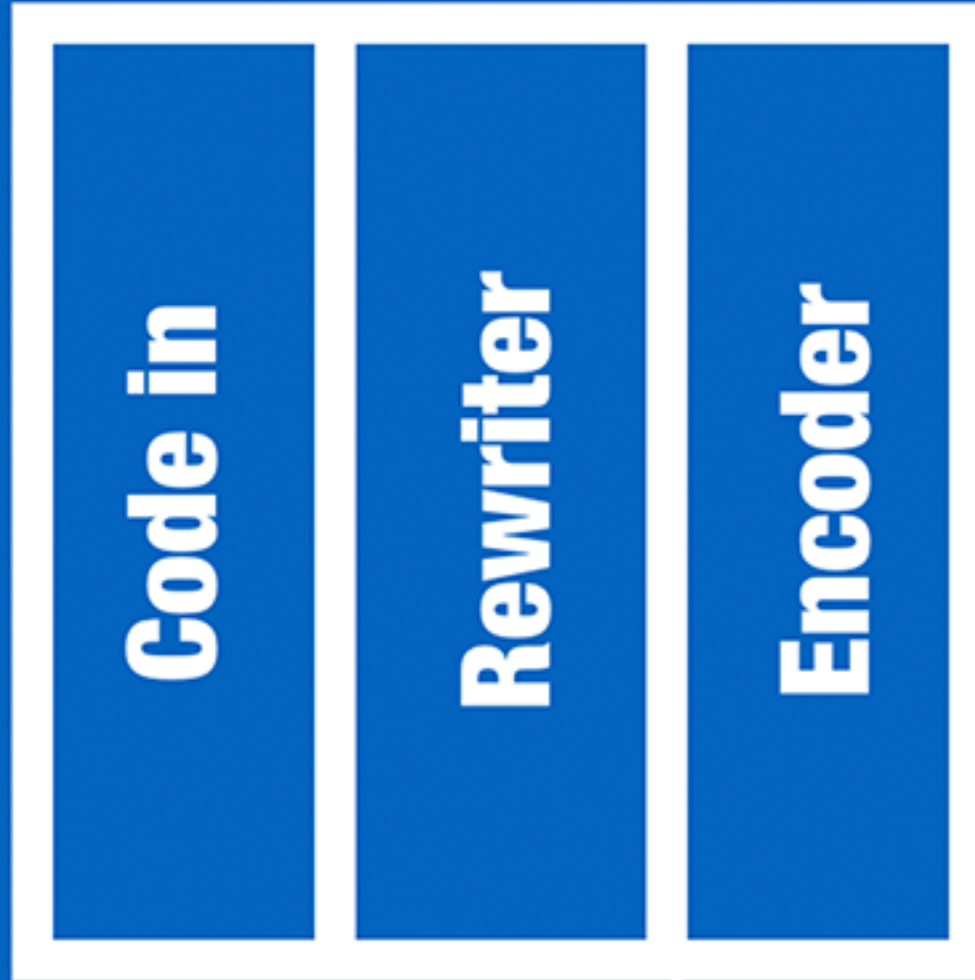


**Optimization
Decision**

How does it work?

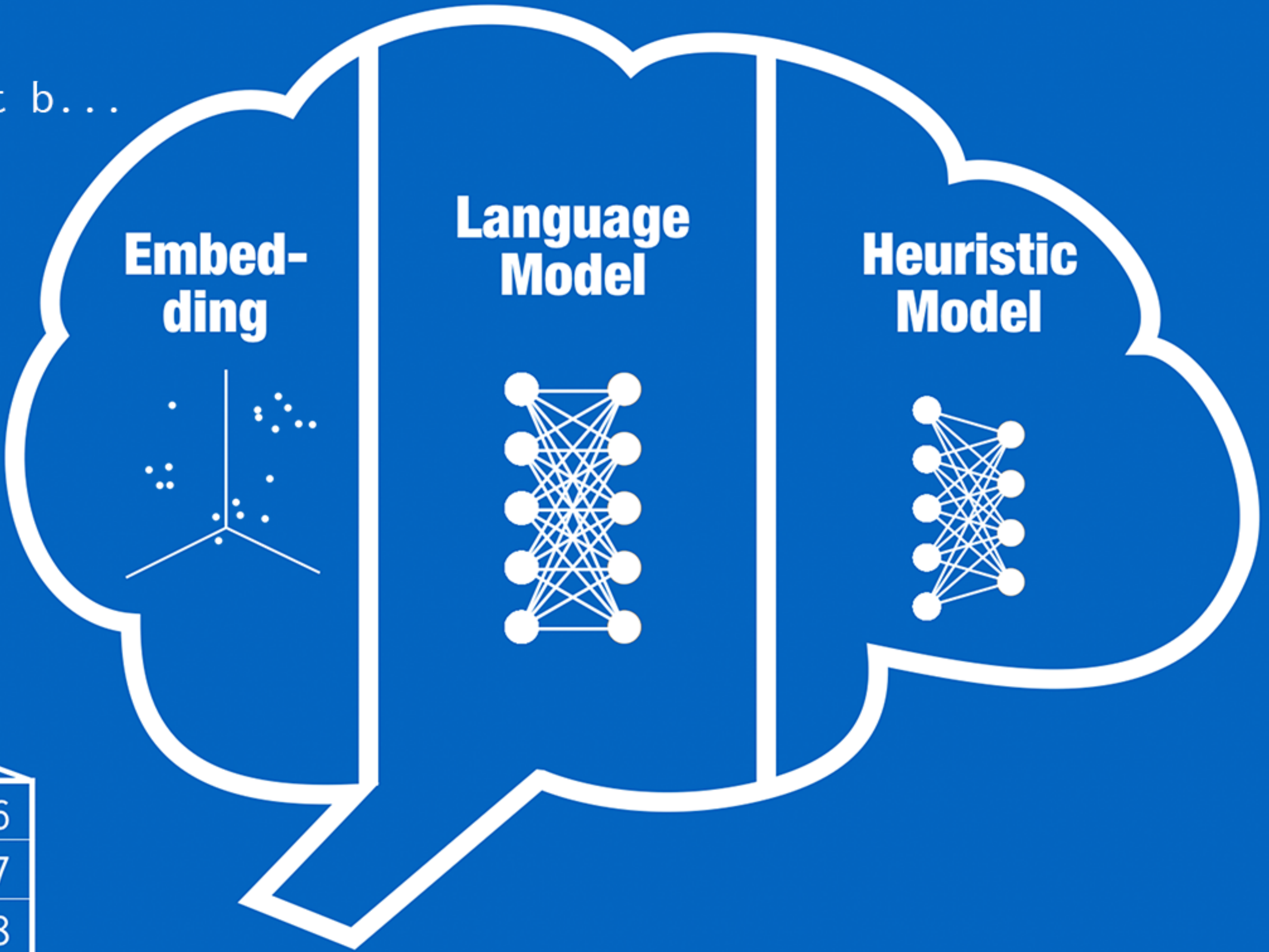
```
kernel void memset_kernel(  
  global char* mem_d, short val...
```

```
kernel void A(  
  global char* a, short b...
```



Vocabulary

_	0	b	6
(1	char	7
*	2	global	8
,	3	kernel	9
A	4	short	10
a	5	void	11



well
How does it work?

Prior Art

Heterogeneous Mapping

Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems

Dominik Grewe Zheng Wang Michael F.P. O'Boyle
School of Informatics, University of Edinburgh
{dominik.grewe, zh.wang}@ed.ac.uk, mob@inf.ed.ac.uk

Abstract

General purpose GPU based systems are highly attractive as they give potentially massive performance at little cost. Realizing such potential is challenging due to the complexity of programming. This paper presents a compiler based approach to automatically generate optimized OpenCL code from data parallel OpenMP programs for GPUs. Such an approach brings together the benefits of a clear high level language (OpenMP) and an emerging standard (OpenCL) for heterogeneous multi-cores. A key feature of our scheme is that it leverages existing transformations, especially data transformations, to improve performance on GPU architectures and uses predictive modeling to automatically determine if it is worthwhile moving the OpenCL code on the GPU or OpenMP code on the multi-core host. We applied our approach to the entire NAS parallel benchmark suite and evaluated it on two distinct GPU based systems: Core i7/NVIDIA GeForce GTX 580 and Core i7/AMD Radeon 7970. We achieved average (up to) speedups of 4.51x and 4.20x (143x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers

General Terms: Experimentation, Languages, Measurement, Performance

Keywords: GPU, OpenCL, Machine-Learning Mapping

1. Introduction

Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive

performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. Users typically have to identify potential sections of their code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier for users.

High level shared memory programming languages such as OpenMP are more attractive. They give a simple upgrade path to parallelism for existing programs using pragmas. Although OpenMP is mainly used for programming shared memory multi-cores, it is a high-level language with little hardware specific information and can be targeted to other platforms. What we would like is the ease of programming of OpenMP with the GPU availability of OpenCL that is then optimized for a particular platform and gracefully adapts to GPU evolution. We deliver this by developing a compiler based approach that automatically generates optimized OpenCL from a subset of OpenMP. This allows the user to continue to use the same programming language, with no modifications, while benefiting automatically from heterogeneous performance.

The first effort in this direction is [17]. Here, the OpenMPC compiler generates CUDA code from OpenMP programs. While promising, there are two significant shortcomings with this approach. Firstly, OpenMPC does not apply data transformations. As shown in this paper data transformation are crucial to achieve good performance on GPUs. Secondly, the programs are always executed on GPUs. While GPUs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission from Permissions@acm.org.
PACT'14, August 26–27, 2014, Pittsburgh, PA, USA.
Copyright 2014 ACM 978-1-60558-621-4/14/08...\$15.00.
http://dx.doi.org/10.1145/2628001.2628007.

CGO'13
Grewe et. al

Thread Coarsening

Automatic Optimization of Thread-Coarsening for Graphics Processors

Alberto Magni Christophe Dubach Michael O'Boyle
School of Informatics School of Informatics School of Informatics
University of Edinburgh University of Edinburgh University of Edinburgh
United Kingdom United Kingdom United Kingdom
a.magni@sms.ed.ac.uk christophe.dubach@ed.ac.uk mob@inf.ed.ac.uk

ABSTRACT

OpenCL has been designed to achieve functional portability across multi-core devices from different vendors. However, the lack of a single cross-target optimising compiler severely limits performance portability of OpenCL programs. Programmers need to manually tune applications for each specific device, preventing effective portability. We target a compiler transformation specific for data-parallel languages: thread-coarsening and show it can improve performance across different GPU devices. We then address the problem of selecting the best value for the coarsening factor parameter, i.e., deciding how many threads to merge together. We experimentally show that this is a hard problem to solve: good configurations are difficult to find and naive coarsening in fact leads to substantial slowdowns. We propose a solution based on a machine-learning model that predicts the best coarsening factor using kernel-function static features. The model automatically specializes to the different architectures considered. We evaluate our approach on 17 benchmarks on four devices: two Nvidia GPUs and two different generations of AMD GPUs. Using our technique, we achieve speedups between 1.11x and 1.31x on average.

1. INTRODUCTION

Graphical Processing Units (GPUs) are widely used for high performance computing. They provide cost-effective parallelism for a wide range of applications. The success of these devices has led to the introduction of a diverse range of architectures from many hardware manufacturers. This has created the need for a common programming language to harness the available parallelism in a portable way. OpenCL is an industry standard language for GPUs that offers program portability across accelerators of different vendors: a single piece of OpenCL code is guaranteed to be executable on many diverse devices.

A uniform language specification, however, still requires programmers to manually optimize kernel code to improve performance on each target architecture. This is a tedious

process, which requires knowledge of hardware behavior, and must be repeated each time the hardware is updated. This problem is particularly acute for GPUs which undergo rapid hardware evolution.

The solution to this problem is a cross-architectural optimizer capable of achieving performance portability. Current proposals for cross-architectural compiler support [21, 34] all involve working on source-to-source transformations. Compiler intermediate representations [6] and ISAs [5] that span across devices of different vendors have still to reach full support.

This paper studies the issue of performance portability focusing on the optimization of the thread-coarsening compiler transformation. Thread coarsening [21, 30, 31] merges together two or more parallel threads, increasing the amount of work performed by a single thread, and reducing the total number of threads instantiated. Selecting the best coarsening factor, i.e., the number of threads to merge together, is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions. Making the correct choice leads to significant speedups on all our platforms. Our data show that picking the optimal coarsening factor is difficult since most configurations lead to performance downgrade and only careful selection of the coarsening factor gives improvements. Selecting the best parameter requires knowledge of the particular hardware platform, i.e., different GPUs have different optimal factors.

In this work we select the coarsening factor using an automated machine learning technique. We build our model based on a cascade of neural networks that decide whether it is beneficial to apply coarsening. The inputs to the model are static code features extracted from the parallel OpenCL code. These features include, among the others, branch divergence and instruction mix information. The technique is applied to four GPU architectures: Fermi and Kepler from Nvidia and Cypress and Tahiti from AMD. While naive coarsening misses optimization opportunities, our approach gives an average performance improvement of 1.16x, 1.11x, 1.33x, 1.31x respectively.

In summary the paper makes the following contributions:

- We provide a characterization of the optimization space across four architectures.
- We develop a machine learning technique based on a neural network to predict coarsening.
- We show significant performance improvements across 17 benchmarks.

PACT'14
Magni et. al

Prior Art

Heterogeneous Mapping

Thread Coarsening

Decision Space

**Binary
classification**
{CPU, GPU}

**One-of-six
classification**
{1, 2, 4, 8, 16, 32}

Model

Decision Tree

Cascading

Neural Networks

Prior Art

Heterogeneous Mapping

4 features

Combined from 7 raw values.

Instruction counts / ratios.

Thread Coarsening

Features

7 features

Principle Components of 34 raw values.

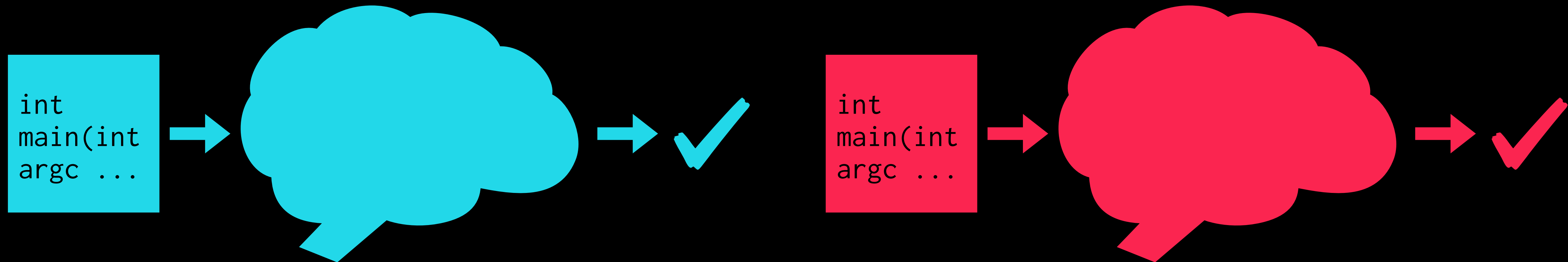
2 papers! 

Instruction counts / ratios / relative deltas.

Our Approach

Heterogeneous Mapping

Thread Coarsening



- 1. Use the same model design for both**
- 2. No tweaking of parameters**
- 3. Minimum change - 3 line diff**

Prior Art

Heterogeneous Mapping

Thread Coarsening

Hardware

**2x CPU-GPU
architectures**

**4x GPU
architectures**

Training Programs

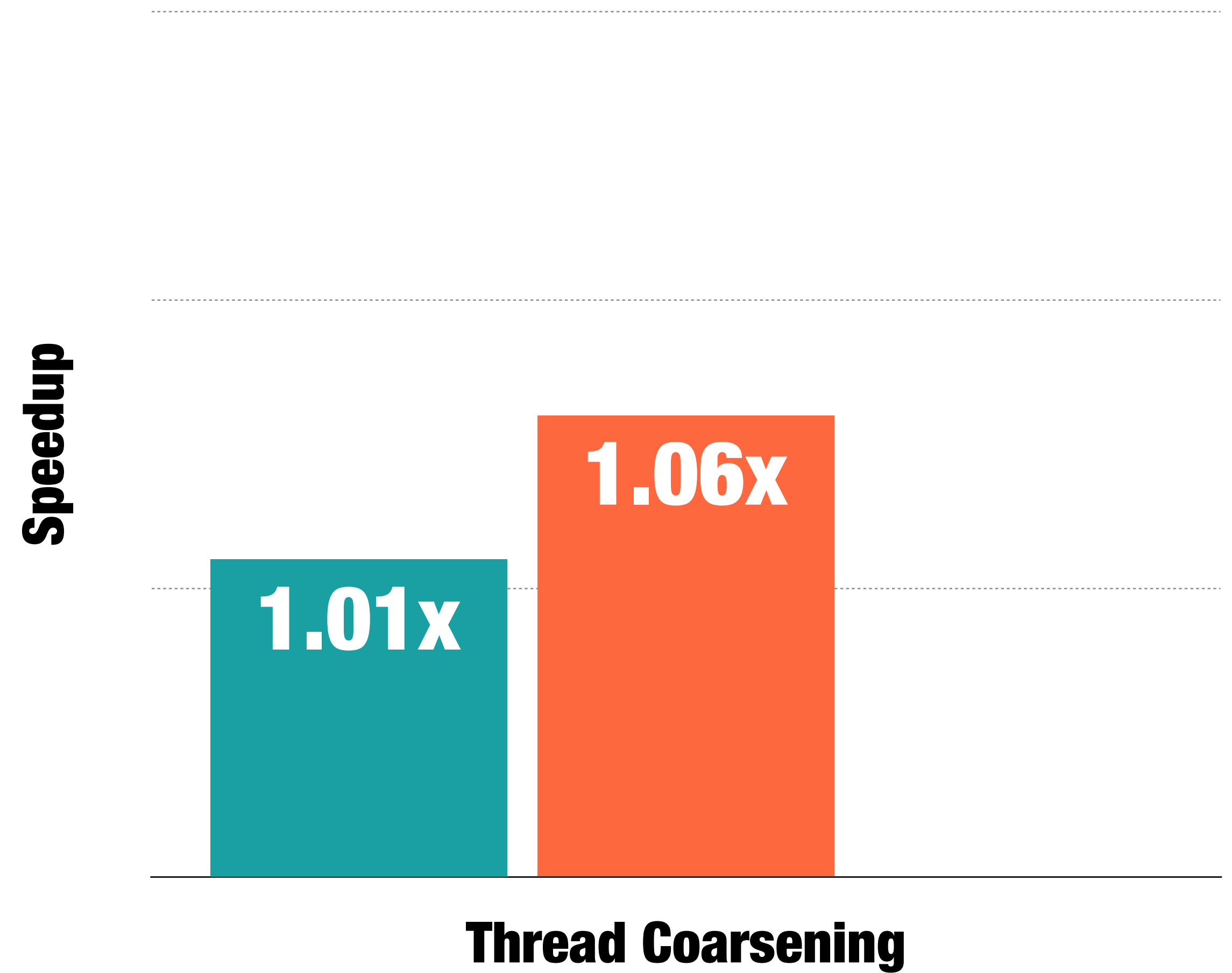
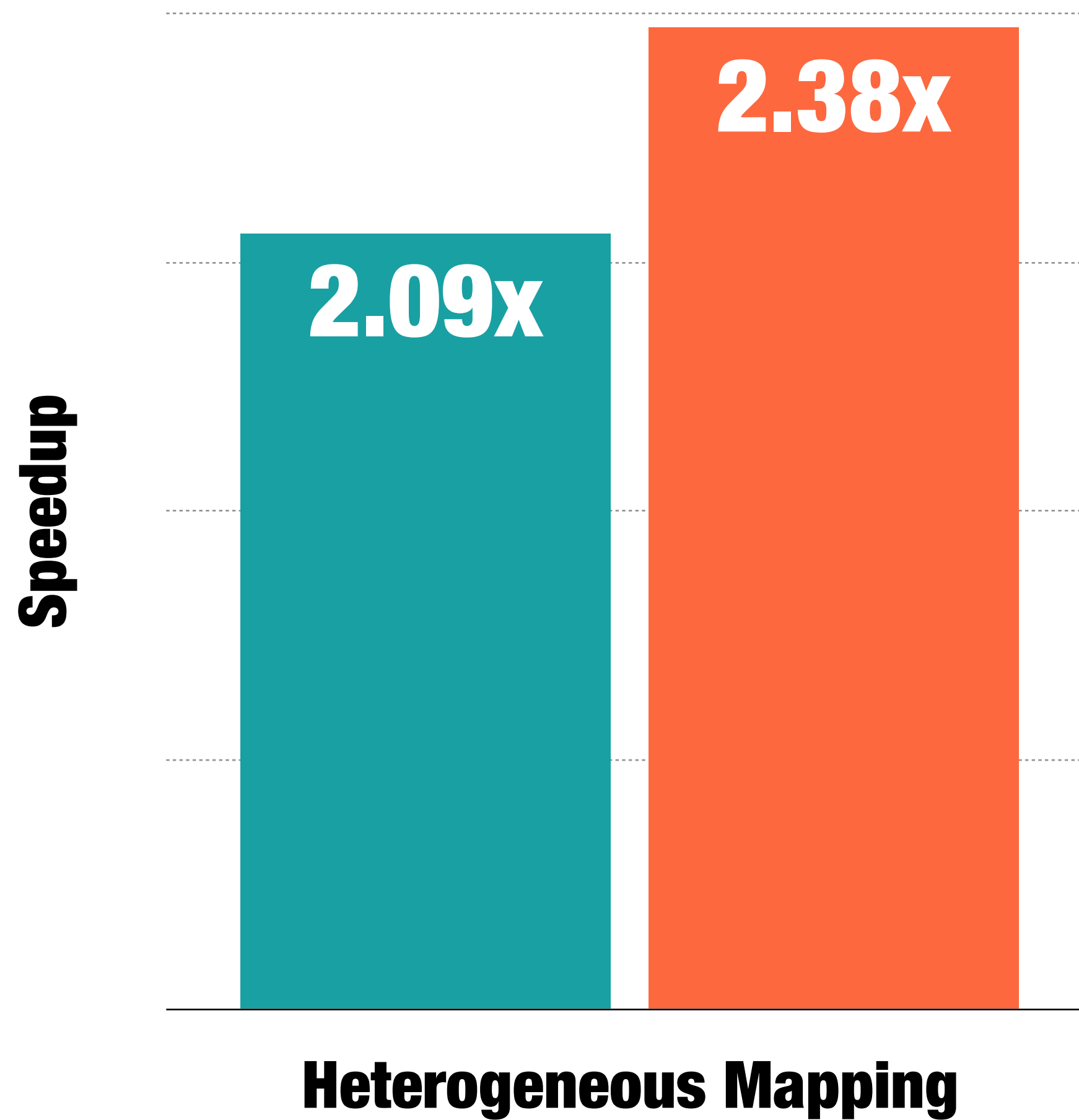
7 Benchmark Suites

3 Benchmark Suites

results

14% and 5% improvements over state-of-the-art

■ **State-of-the-art** ■ **DeepTune**

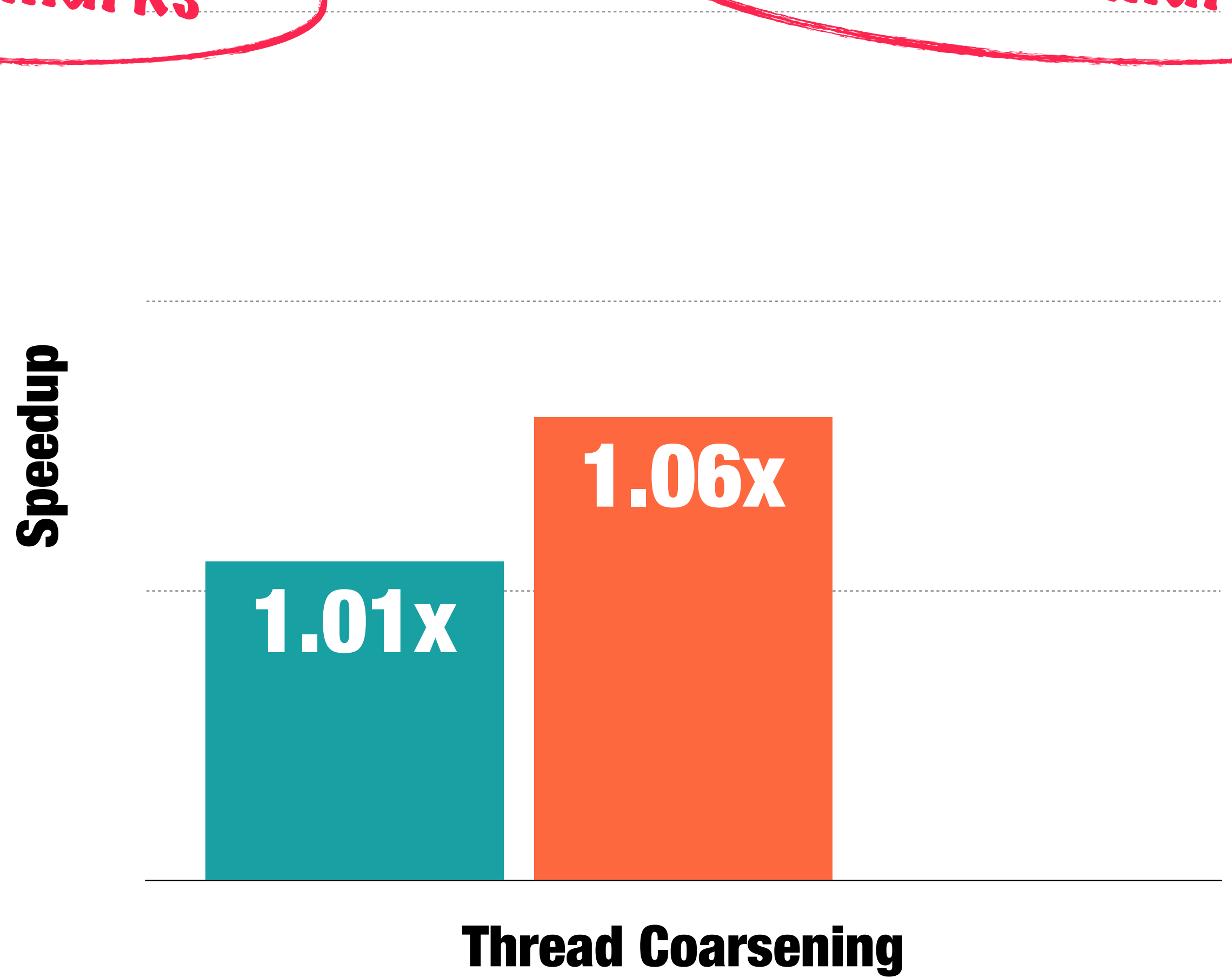
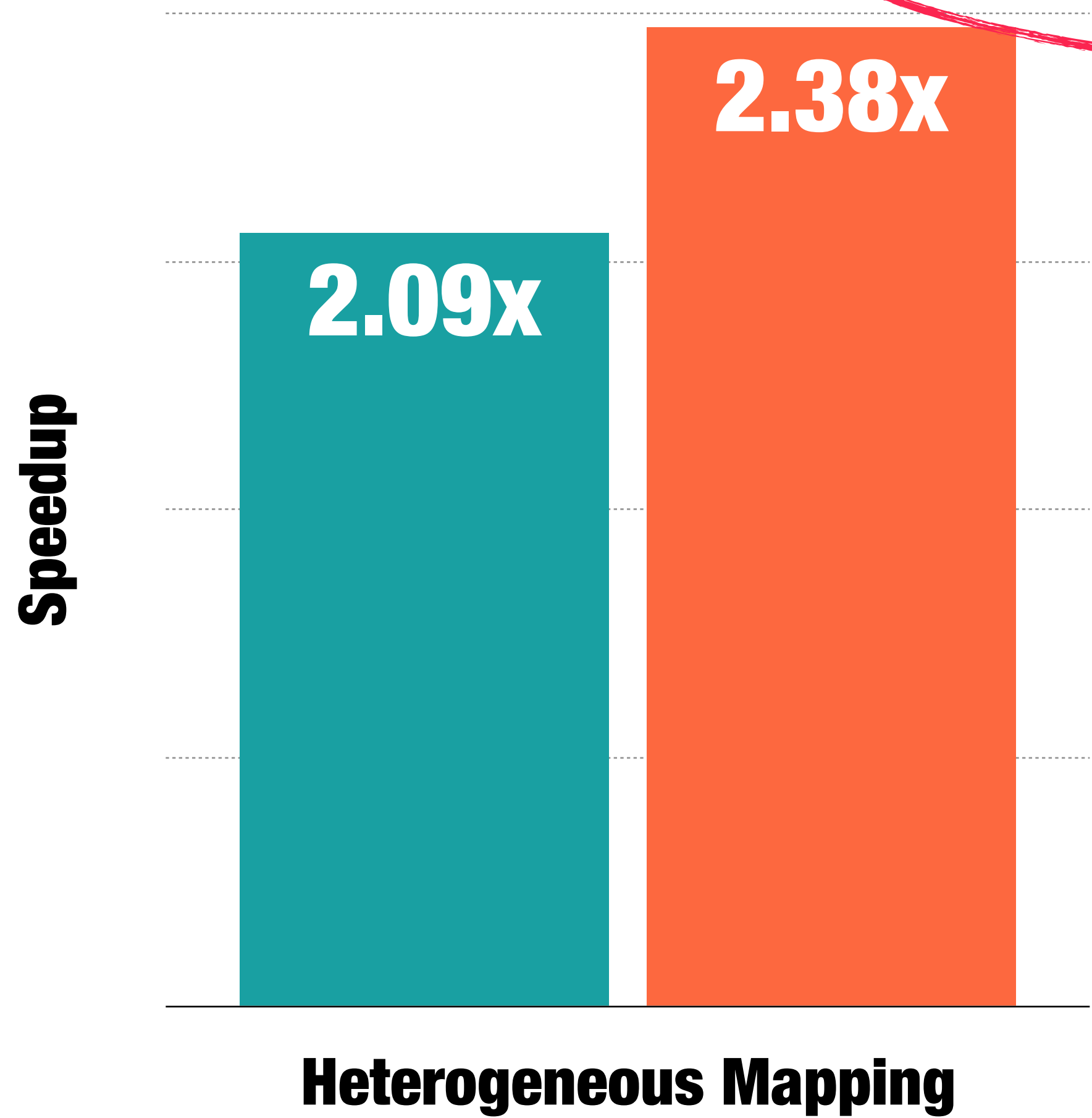


14% and 5% improvements over state-of-the-art

■ State-of-the-art ■ DeepTune

256 benchmarks

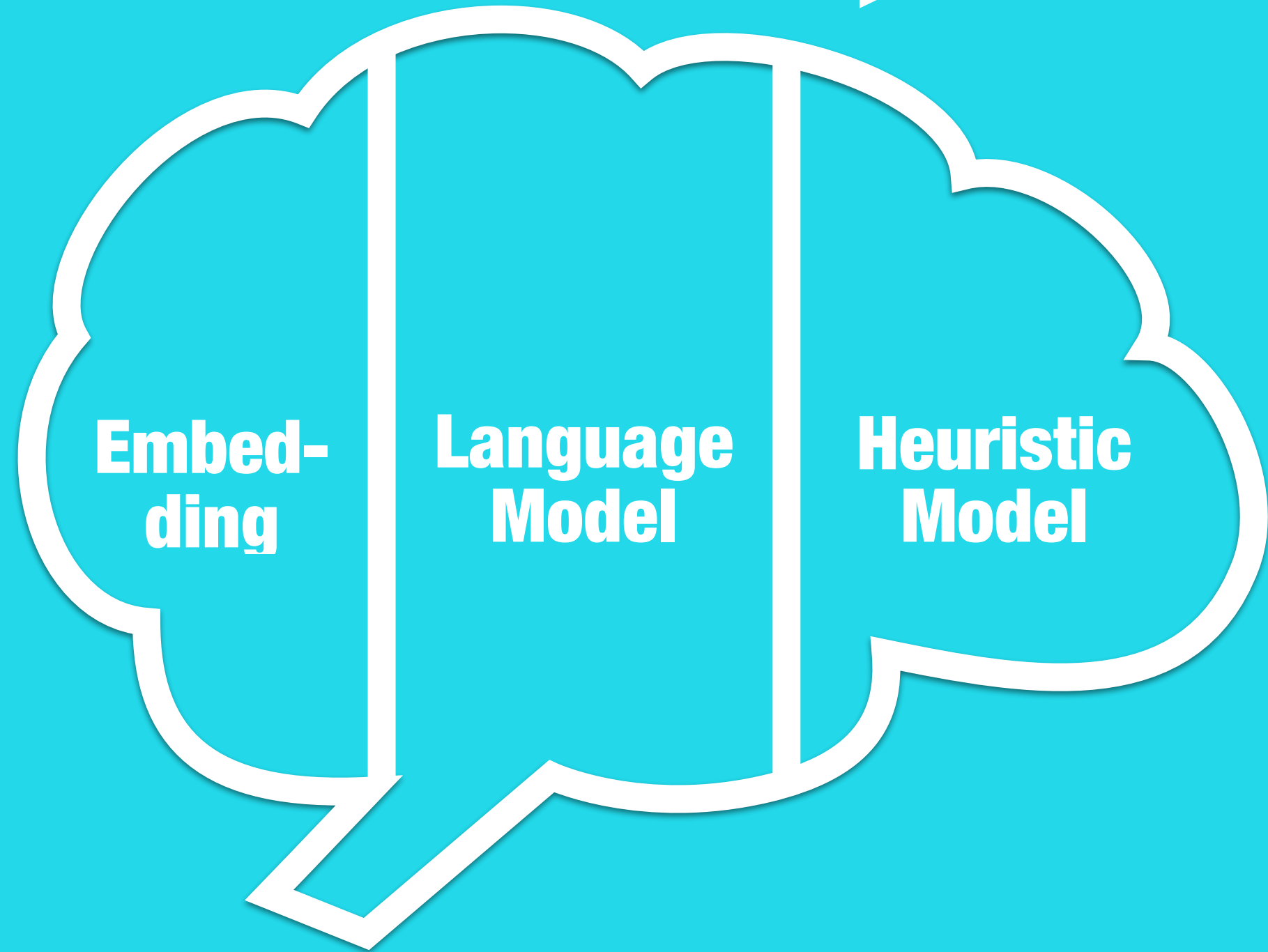
17 benchmarks



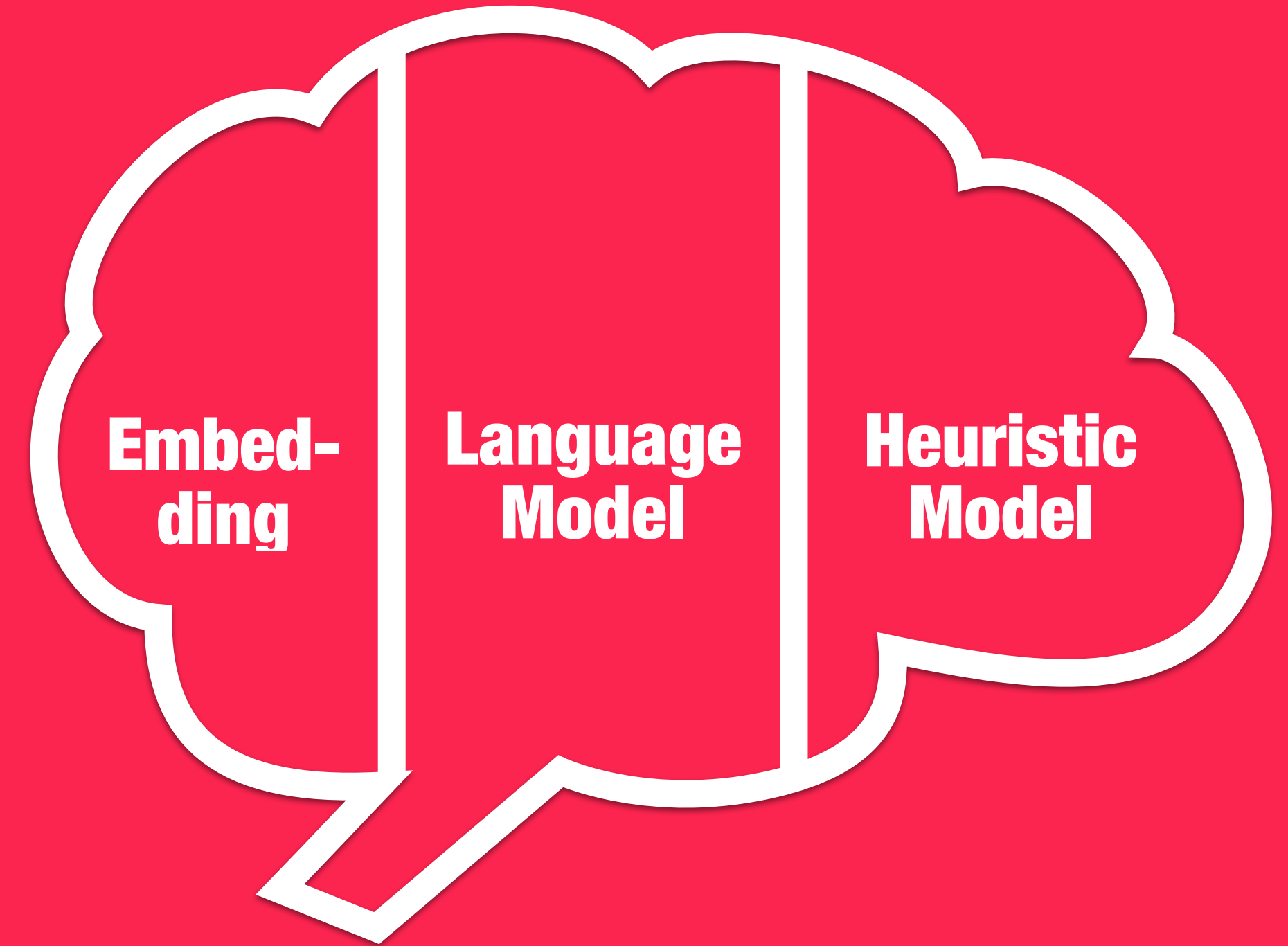
Transfer Learning

Heterogeneous Mapping

general  specialized



Thread Coarsening

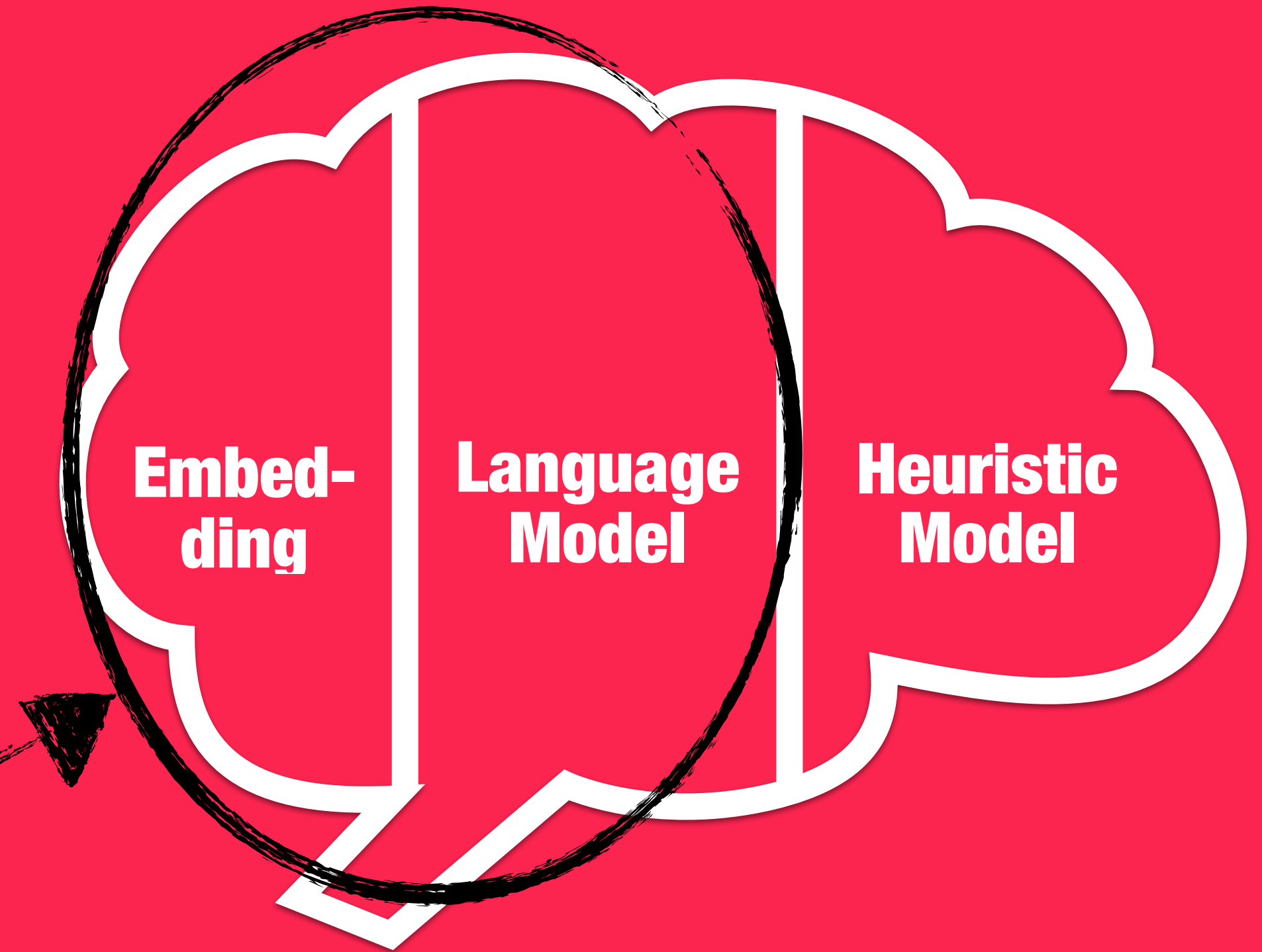
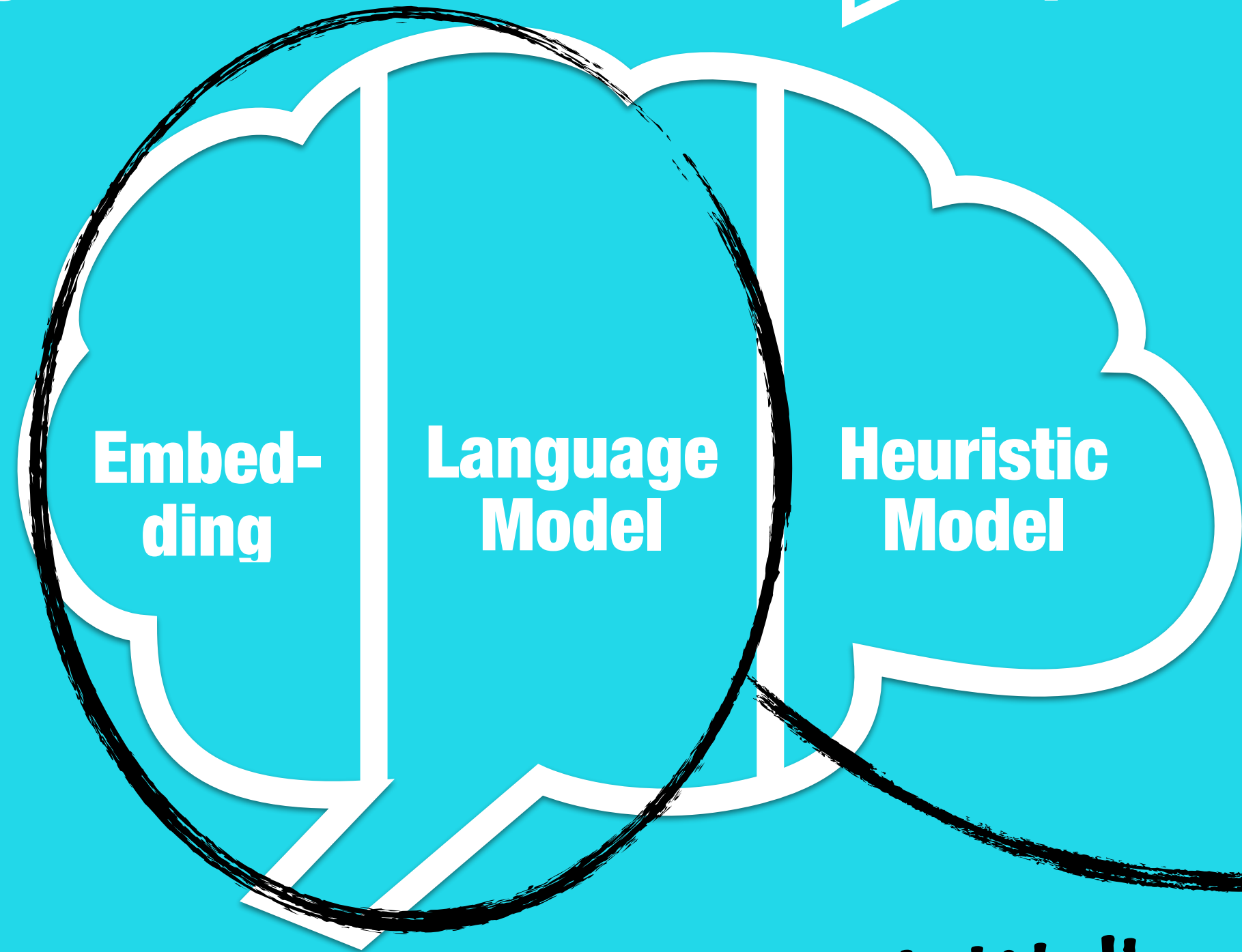


Transfer Learning

Heterogeneous Mapping

Thread Coarsening

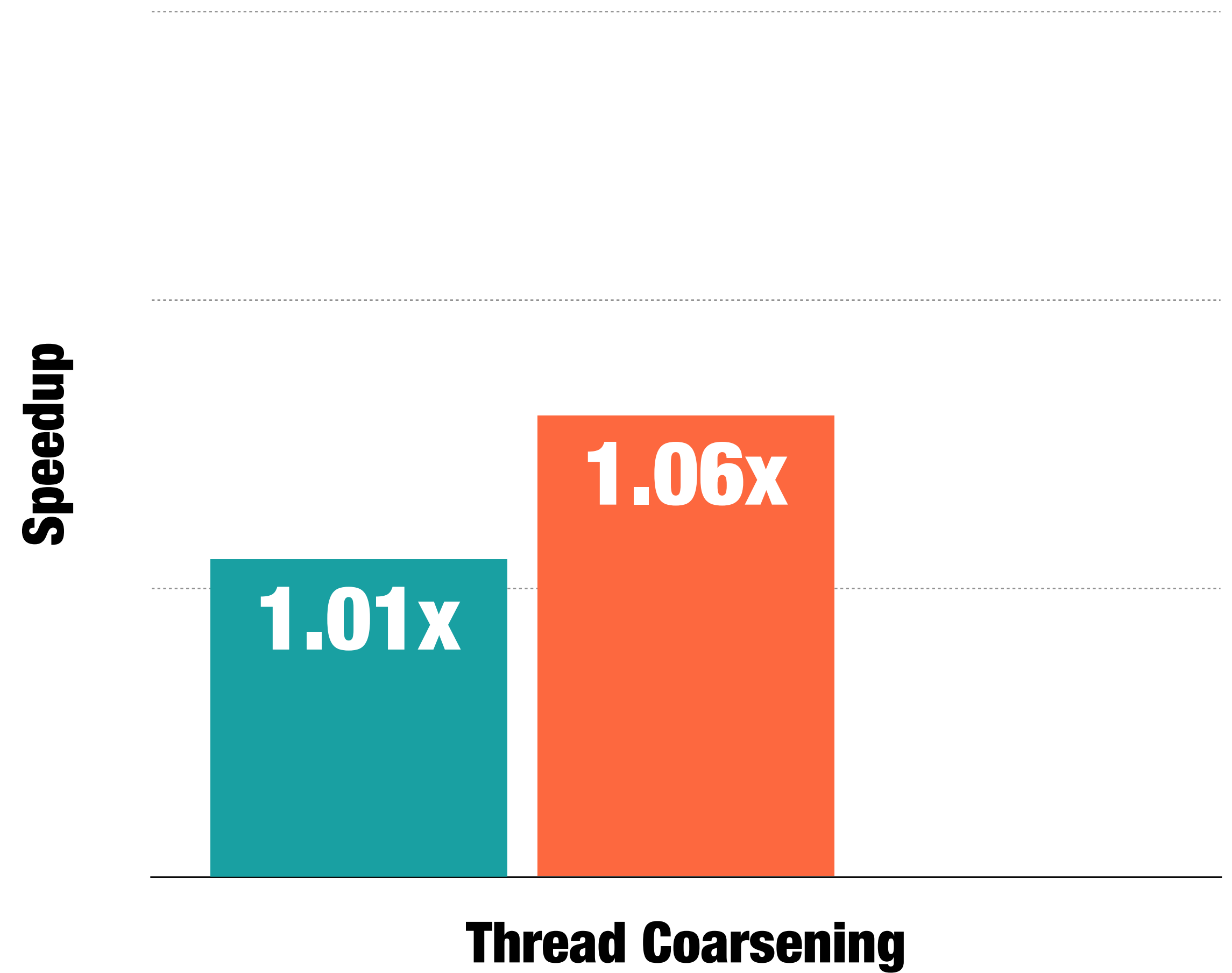
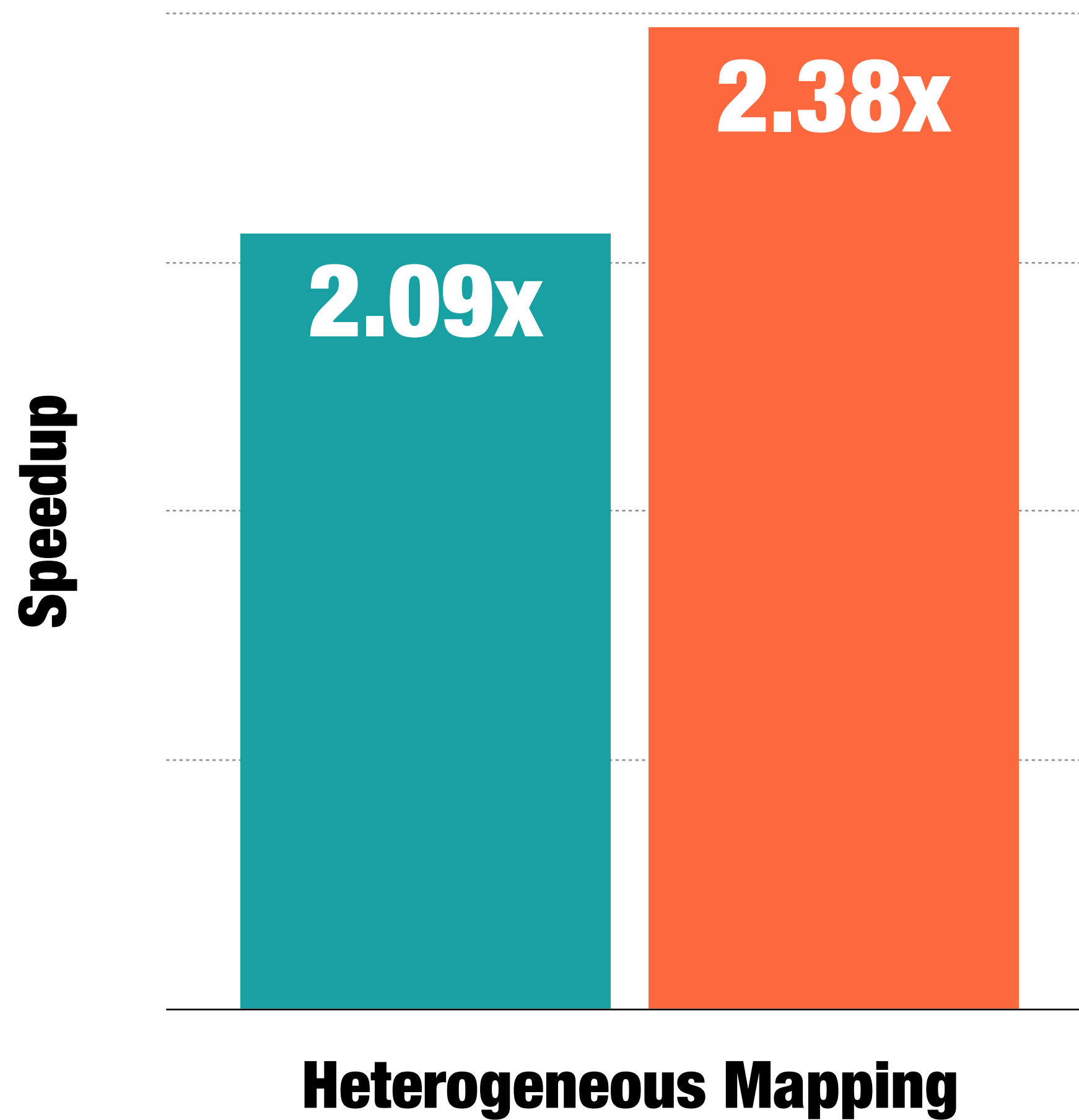
general  specialized



initialize with values

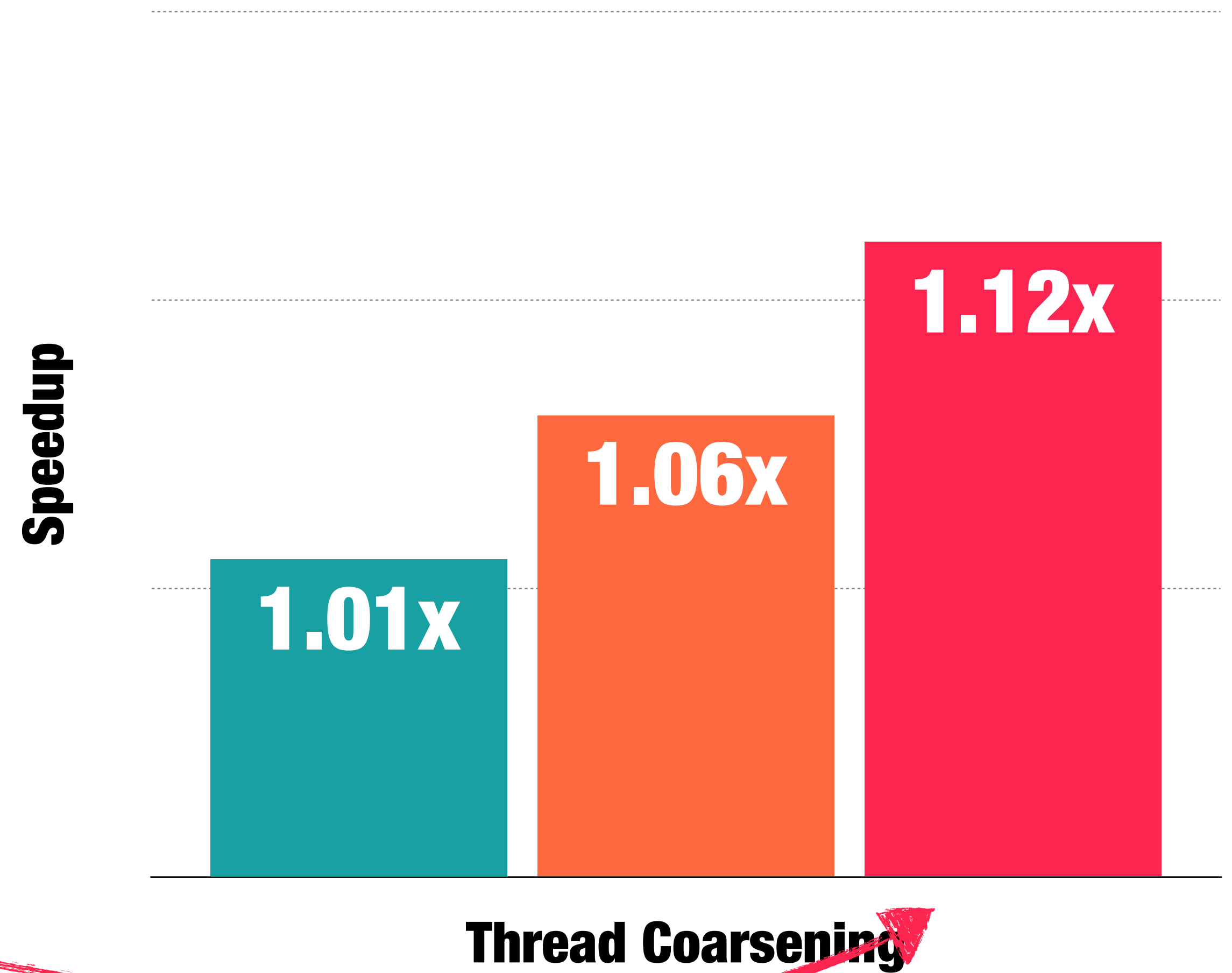
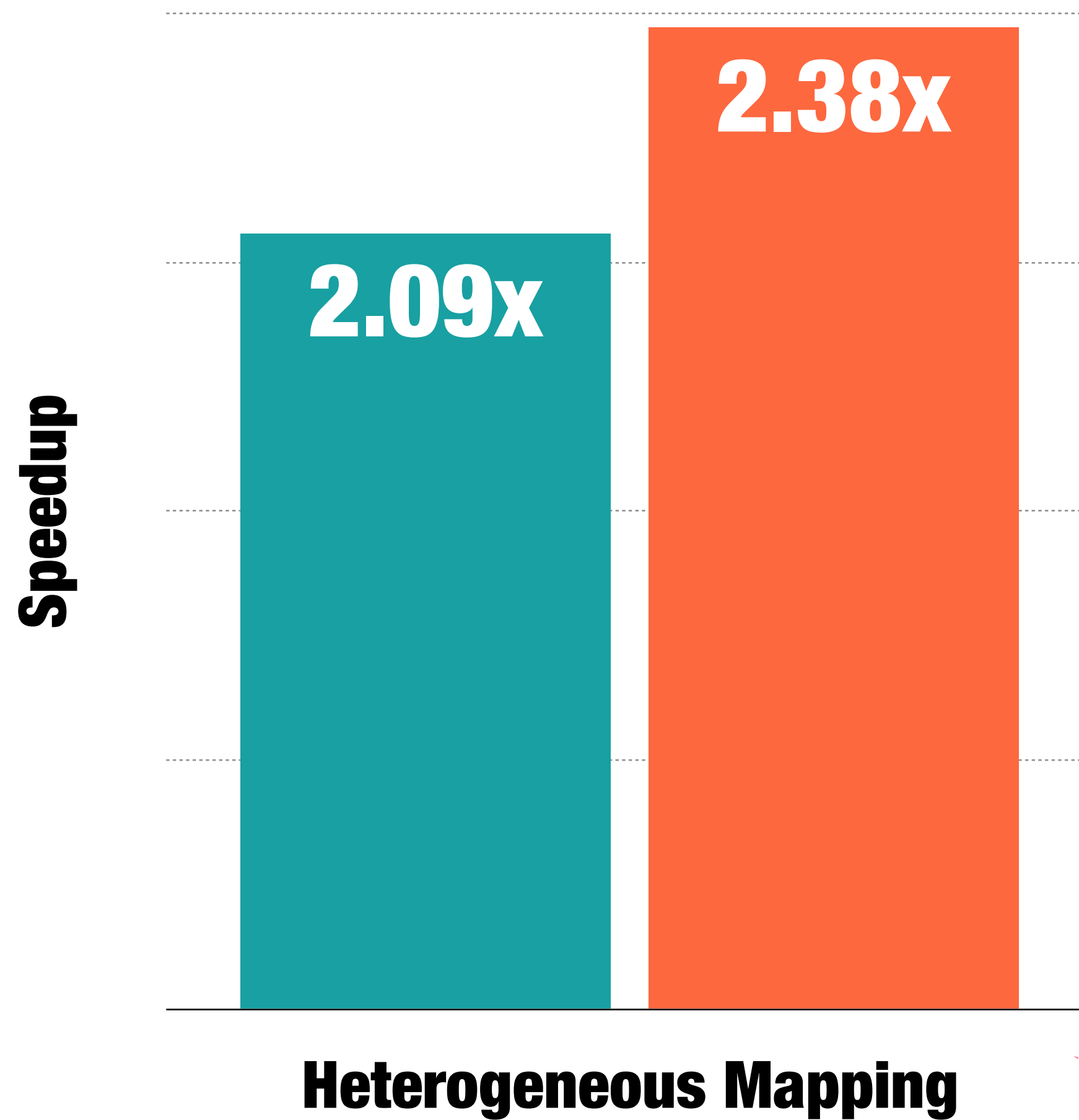
14% and 5% improvements over state-of-the-art

■ **State-of-the-art** ■ **DeepTune**



14% and 11% improvements over state-of-the-art

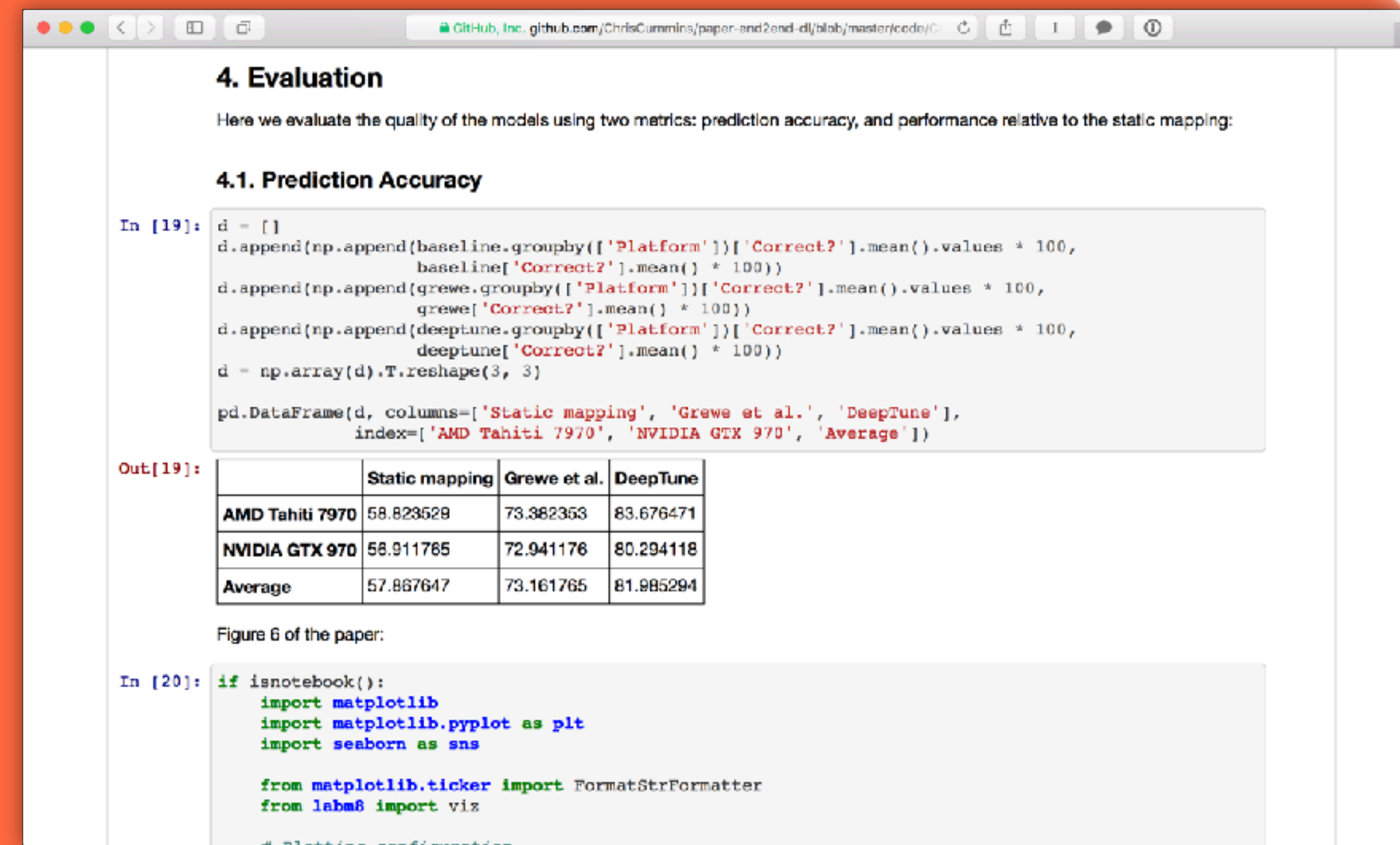
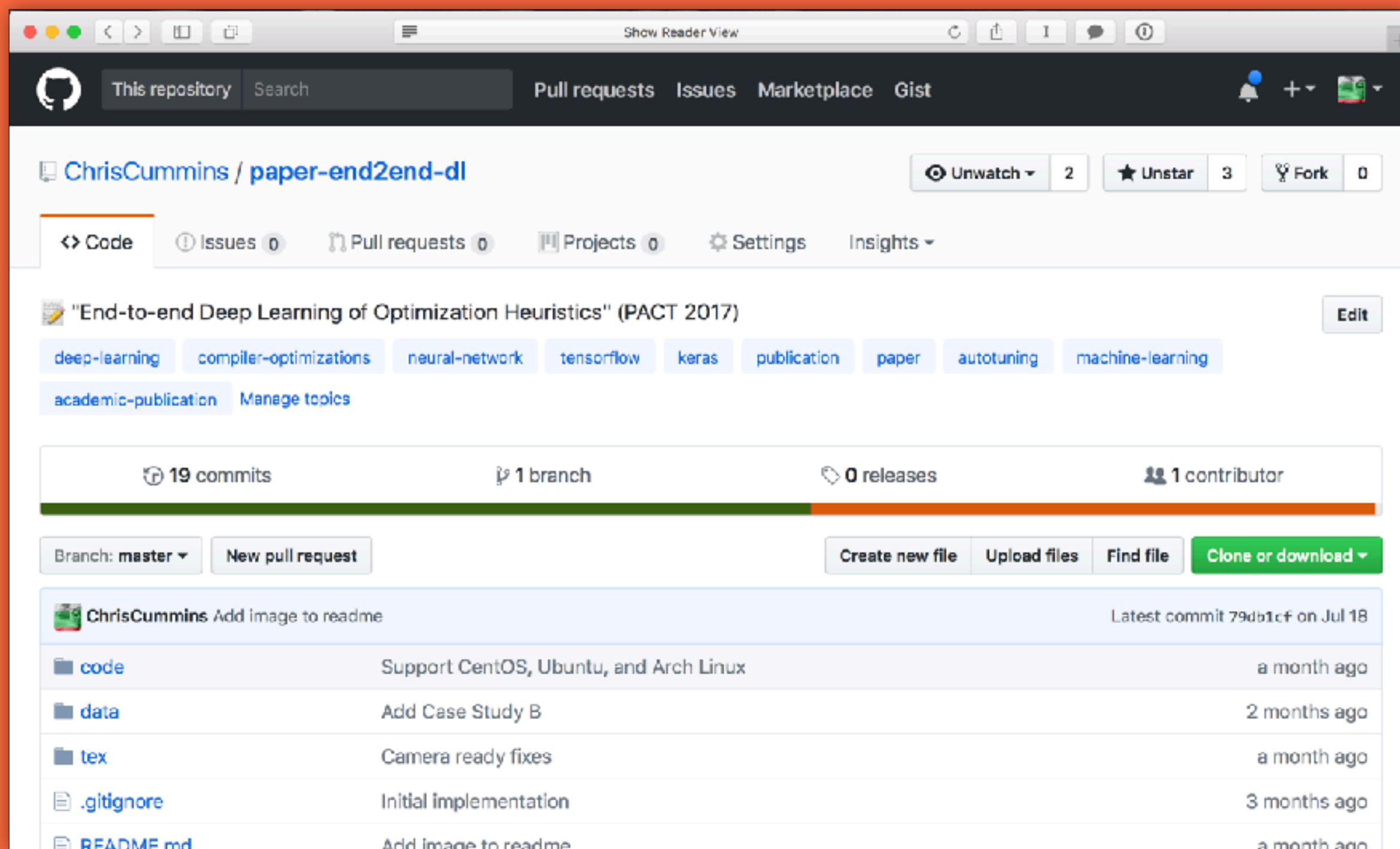
■ **State-of-the-art** ■ **DeepTune** ■ **w. Transfer Learning**



Try it for yourself!



Fork me on GitHub



code and data on GitHub

runs in the browser

<http://chriscummins.cc/pact17>

End-to-end

Deep Learning of Optimisation Heuristics

Problem: feature design is hard

Featureless heuristics

First cross-domain learning

11-14% speedups

<http://chriscummins.cc/pact17>