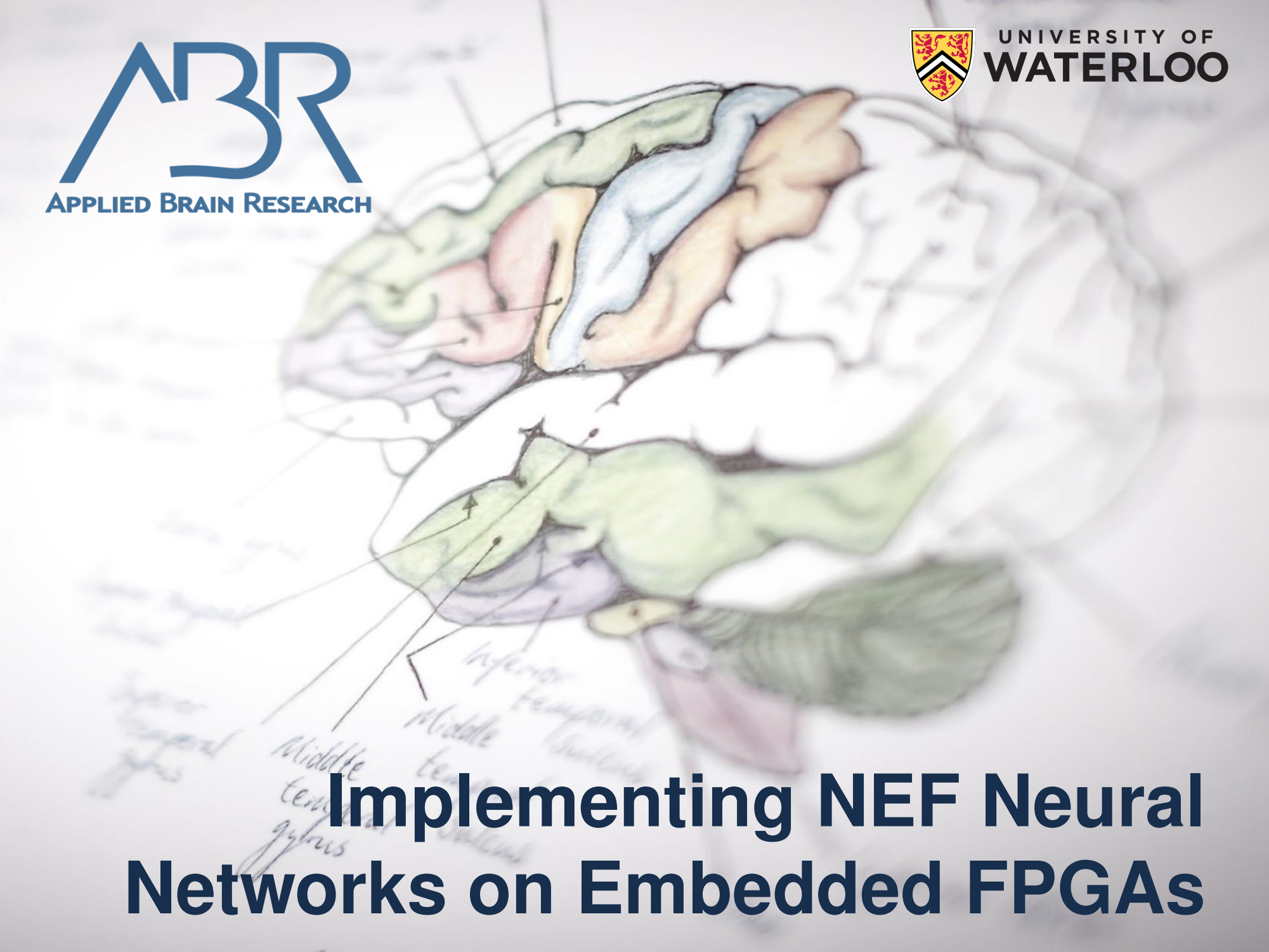


Implementing NEF Neural Networks on Embedded FPGAs

Claim

We develop a Python accessible PYNQ implementation that outperforms the Jetson TX1 GPU computing NEF style neural networks by 10–500$\times$ using 2–10$\times$ less power.

Claim

We develop a Python accessible PYNQ implementation that outperforms the Jetson TX1 GPU computing NEF style neural networks by 10–500× using 2–10× less power.

Our design:

- Is built for embedded applications
- Is built with High-Level Synthesis
- Automatically tunes fixed-point precision

Motivation

- Machine learning is growing

Motivation

- Machine learning is growing
  - ↳ Typically done in Python

Motivation

- Machine learning is growing
  - ↳ Typically done in Python
  - ↳ Compute heavy

Motivation

- Machine learning is growing
  - ↳ Typically done in Python
  - ↳ Compute heavy
- Moore's law is ending

Motivation

- Machine learning is growing
  - ↳ Typically done in Python
  - ↳ Compute heavy
- Moore's law is ending
  - ↳ Want efficient/custom hardware

Context

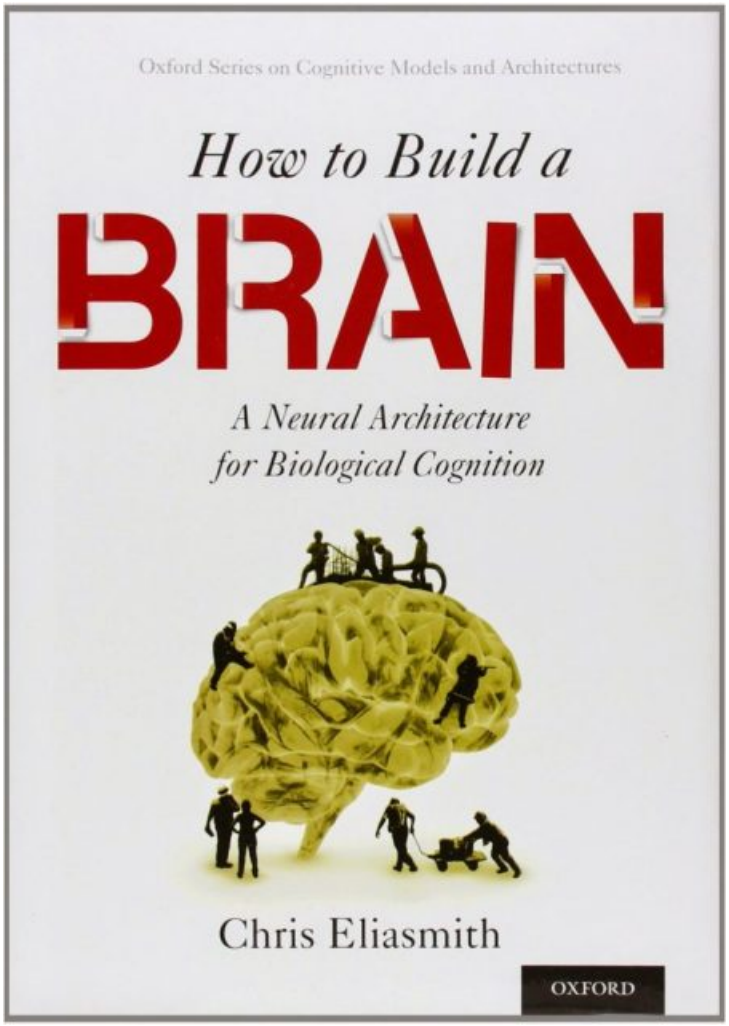
We target the Neural Engineering Framework (NEF) via the Nengo Python package

We target the Neural Engineering Framework (NEF) via the Nengo Python package

- Note NEF has different structure than DNN

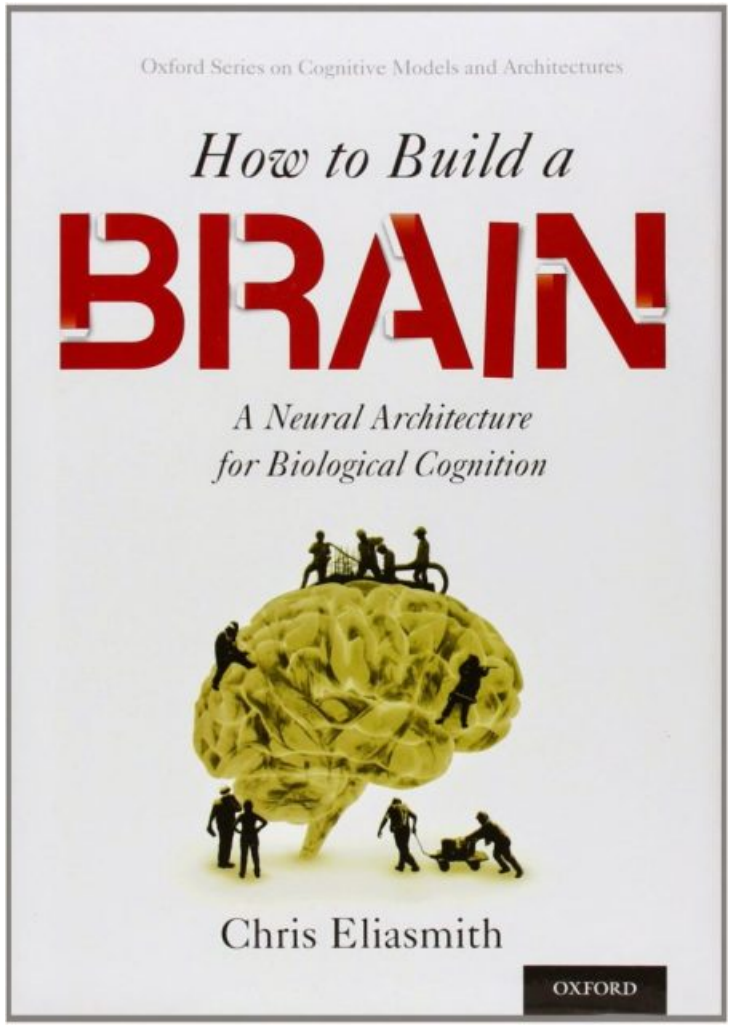
Nengo goes beyond DNN

- Dynamic models



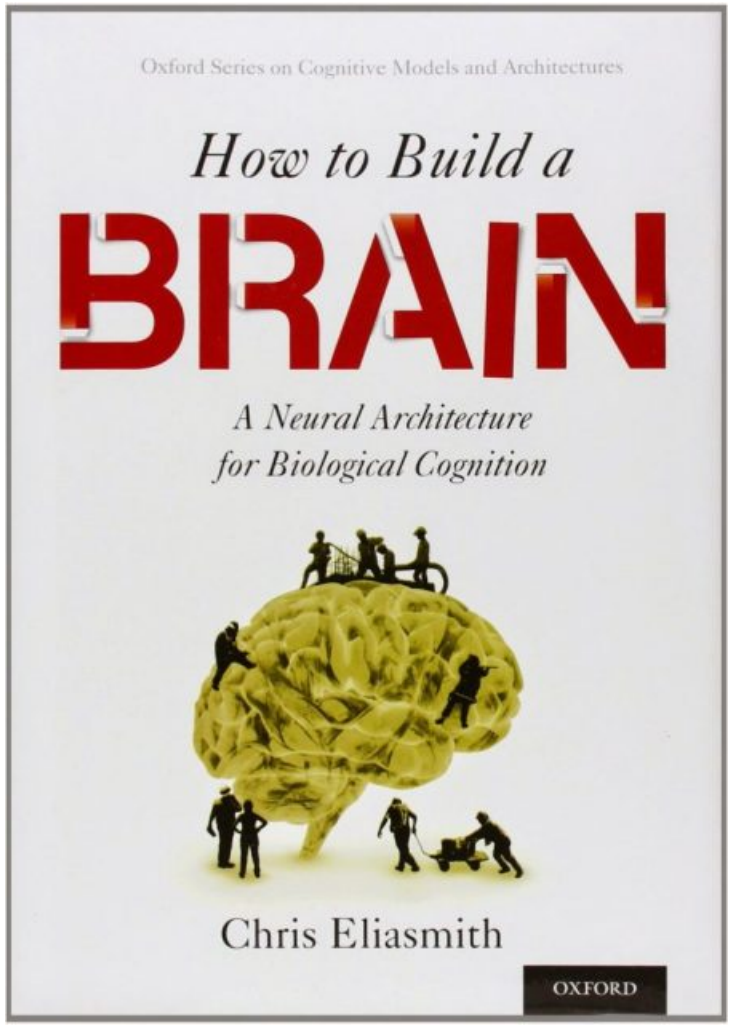
Nengo goes beyond DNN

- Dynamic models
- Real-time adaptive control



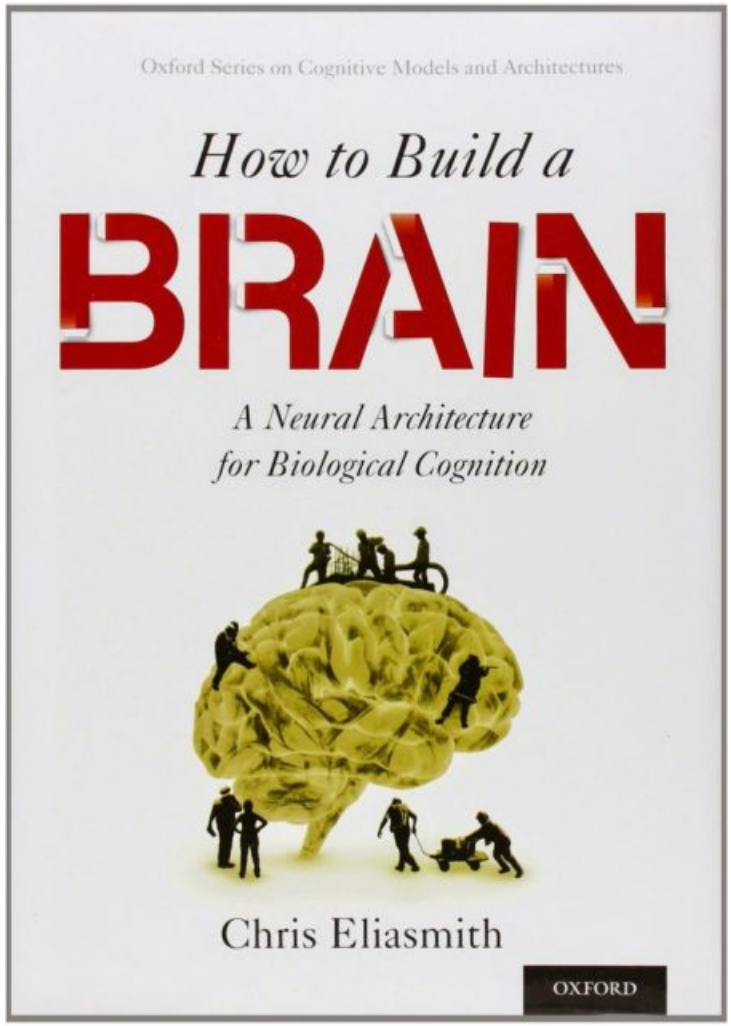
Nengo goes beyond DNN

- Dynamic models
- Real-time adaptive control
- Biologically plausible cognitive architectures

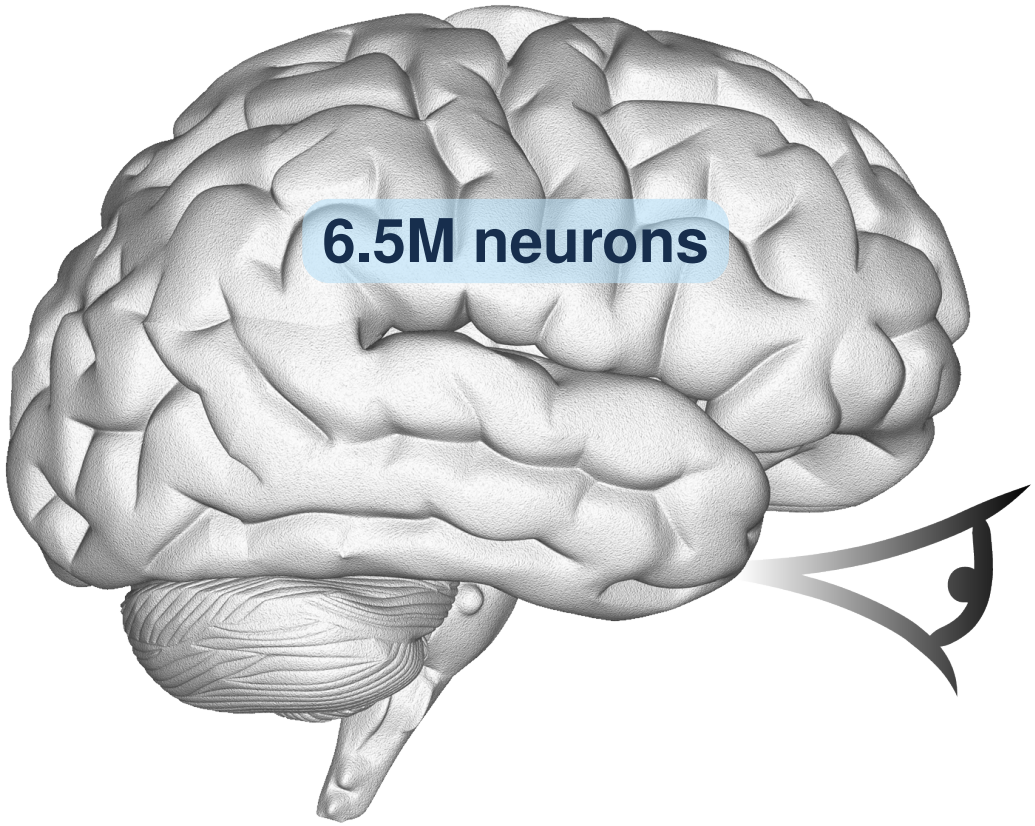


Nengo goes beyond DNN

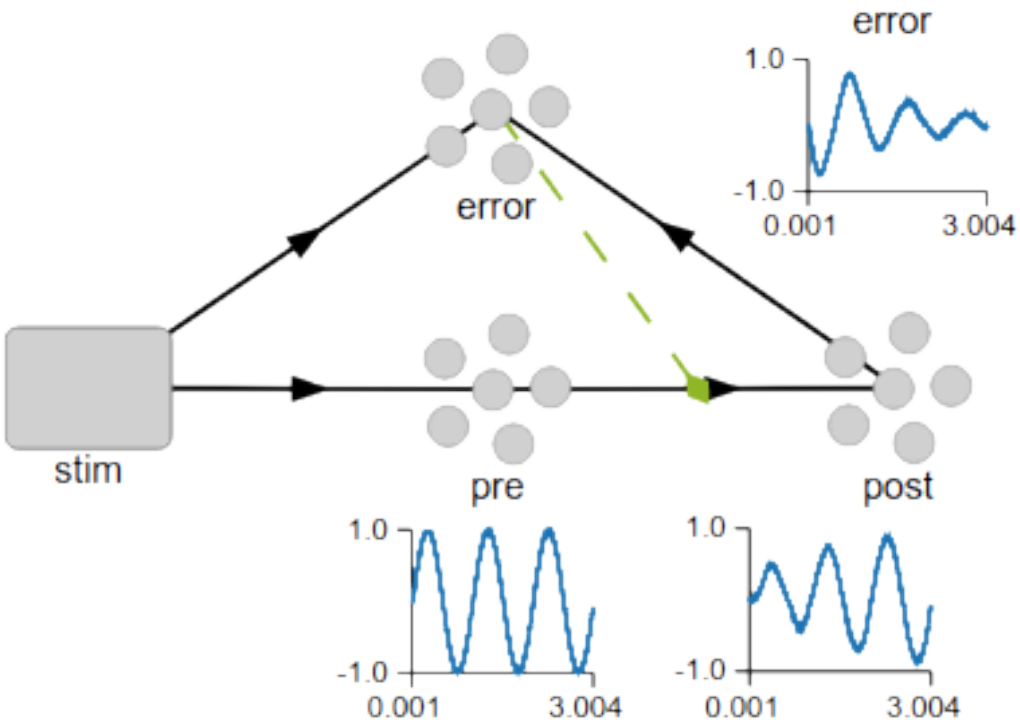
- Dynamic models
- Real-time adaptive control
- Biologically plausible cognitive architectures
- Hierarchical reinforcement learning



The world's largest functional brain model, Spaun, is built with the NEF and Nengo

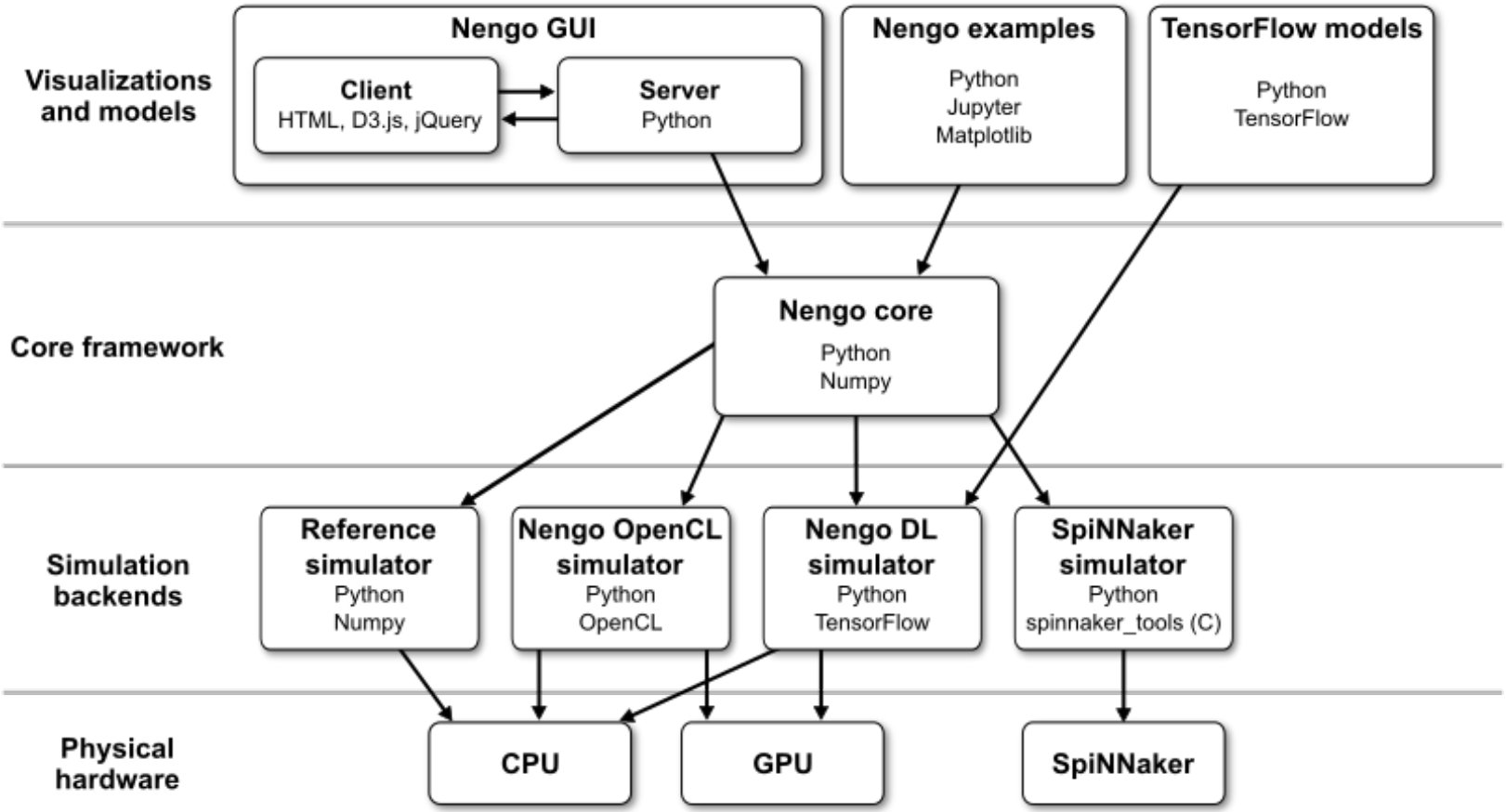


Online learning displayed with Nengo GUI



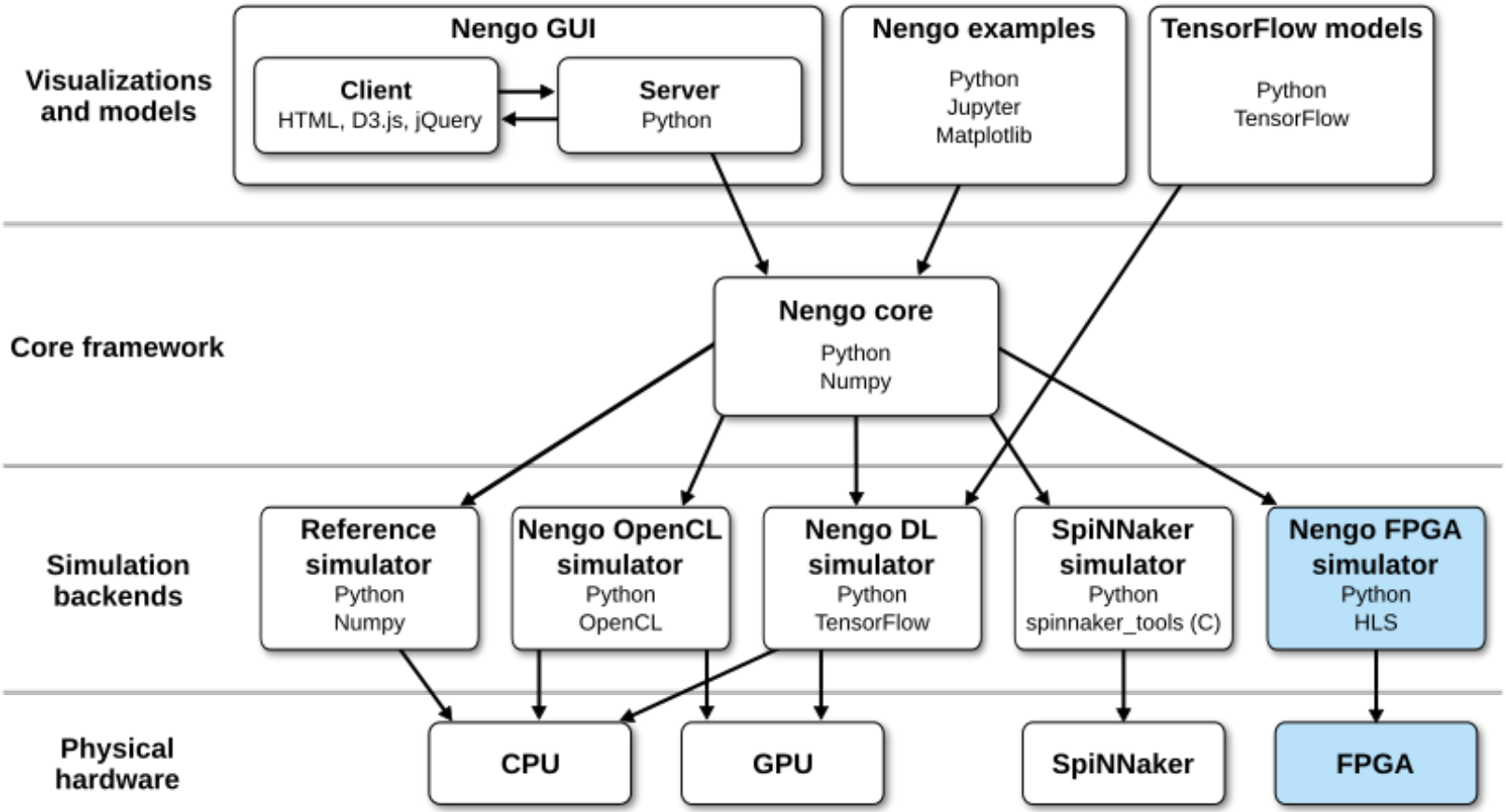
Context

Nengo Ecosystem

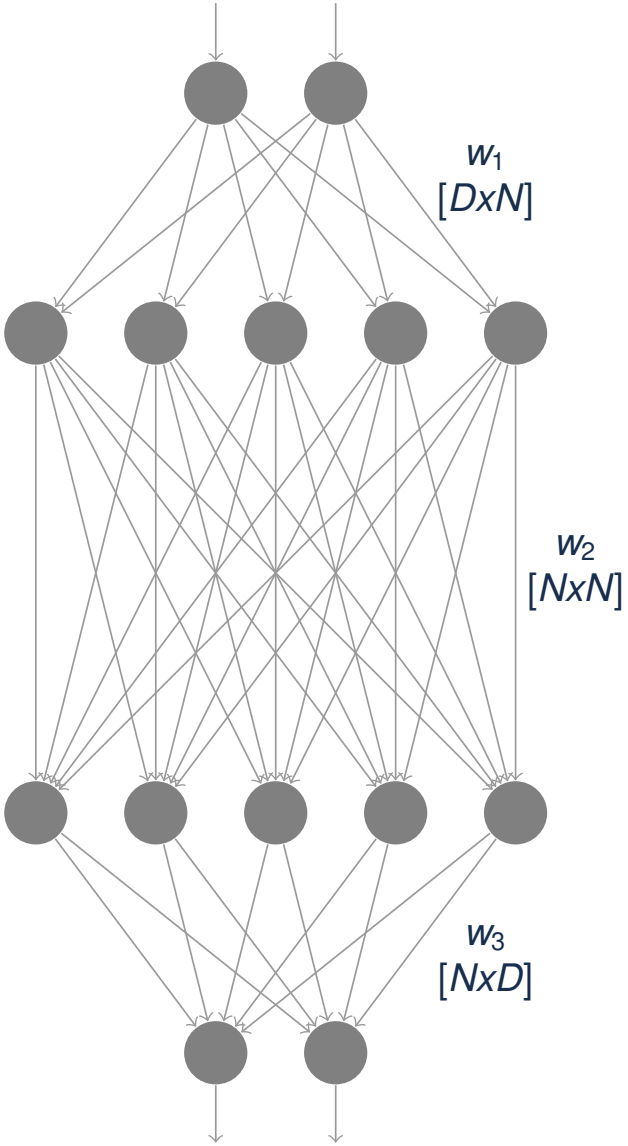


Context

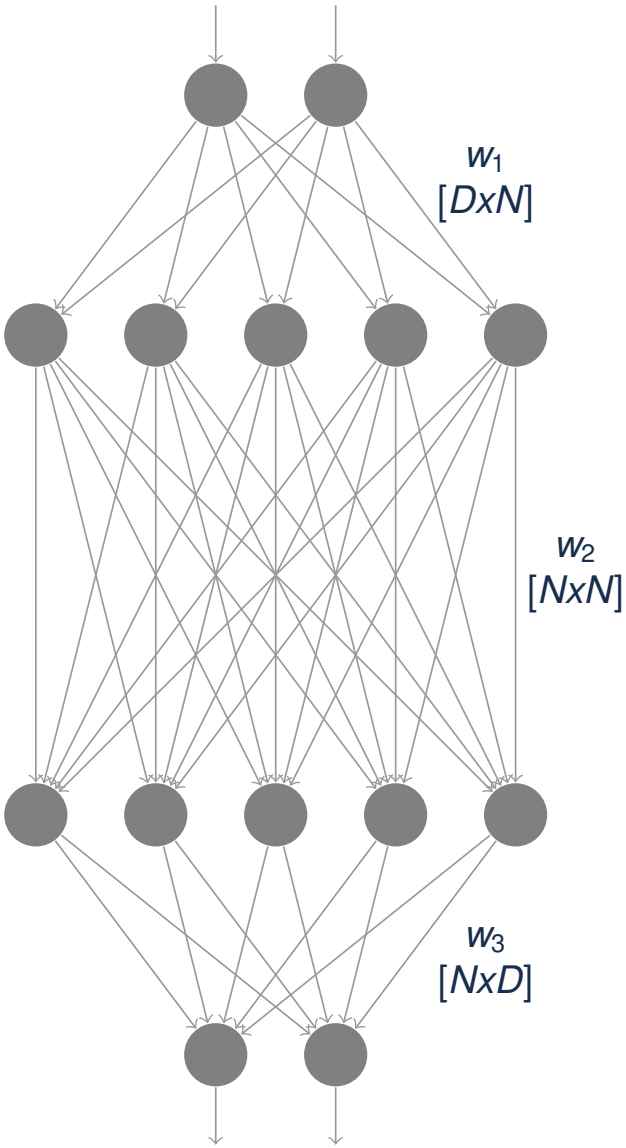
Nengo Ecosystem

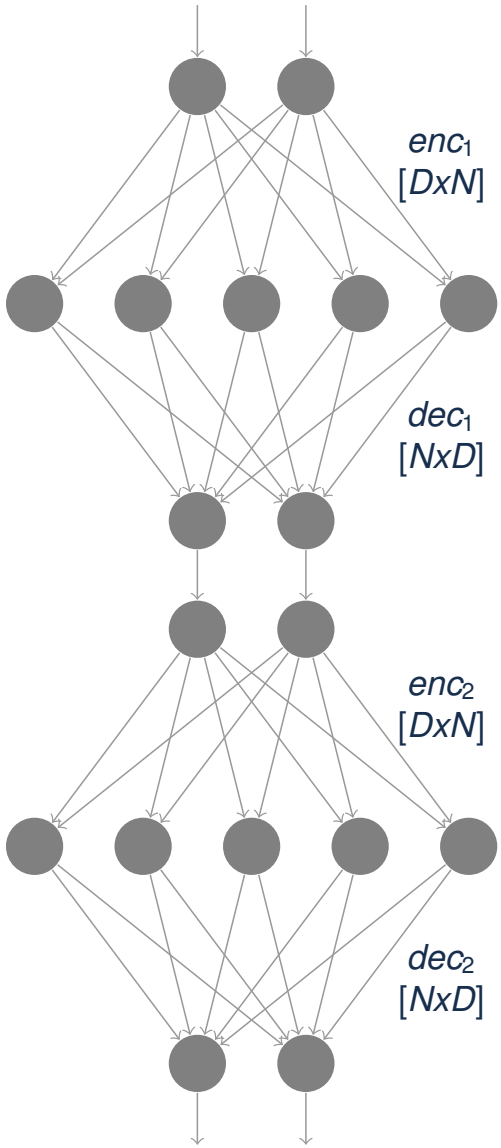


First let's look at typical DNN

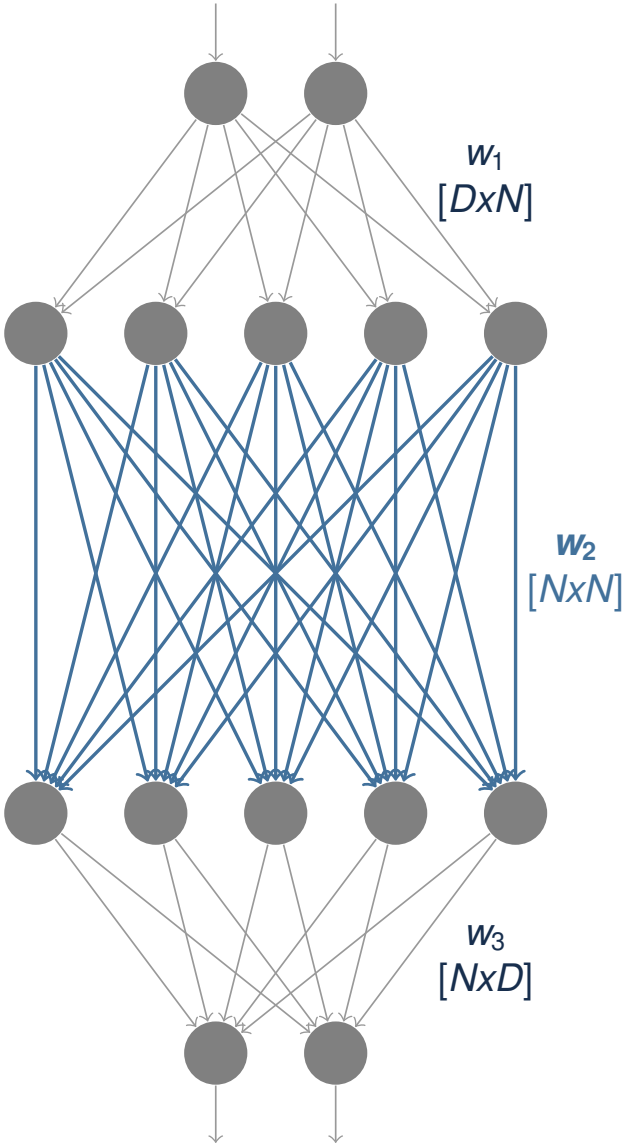


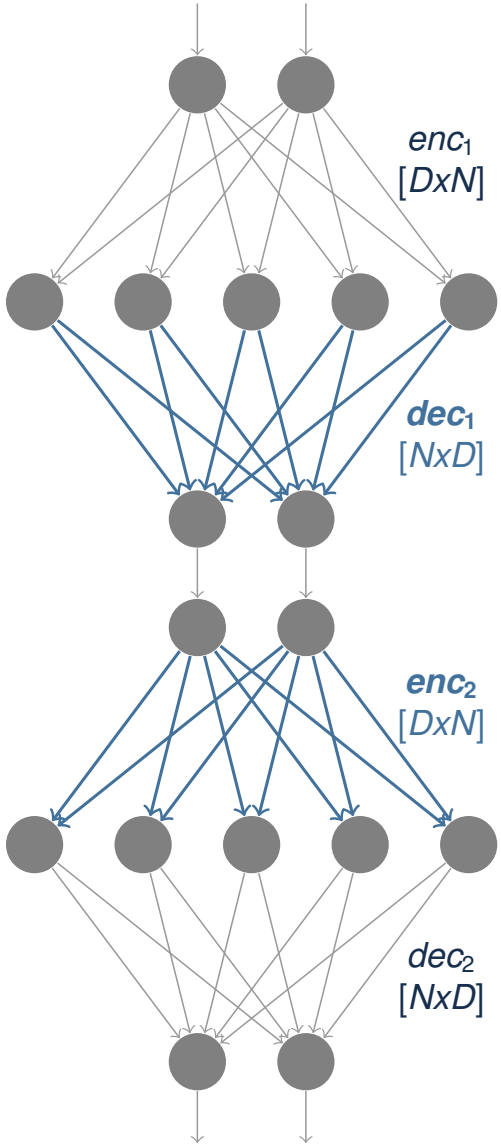
First let's look at typical DNN compared to an NEF network



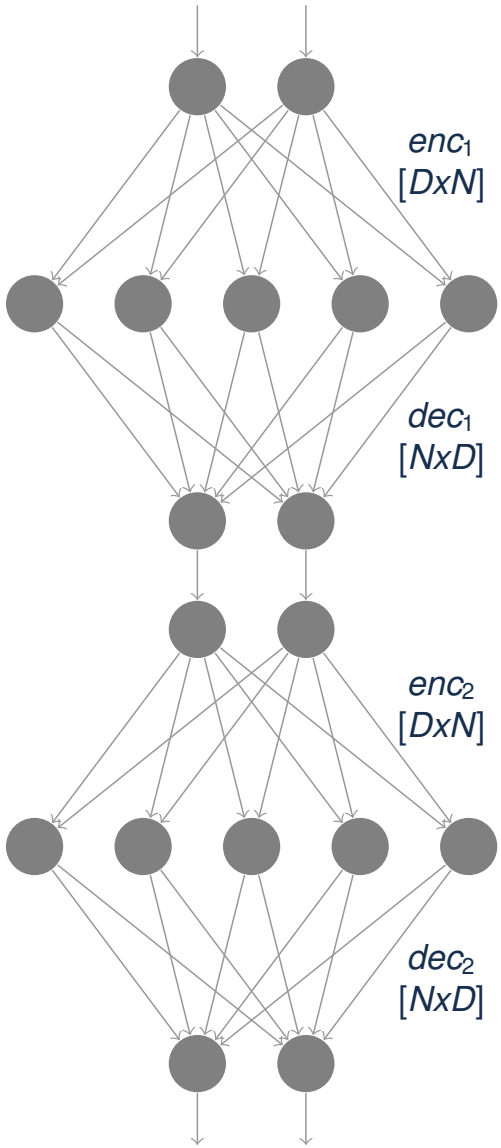


We factor the weight matrix into encode and decode matrices

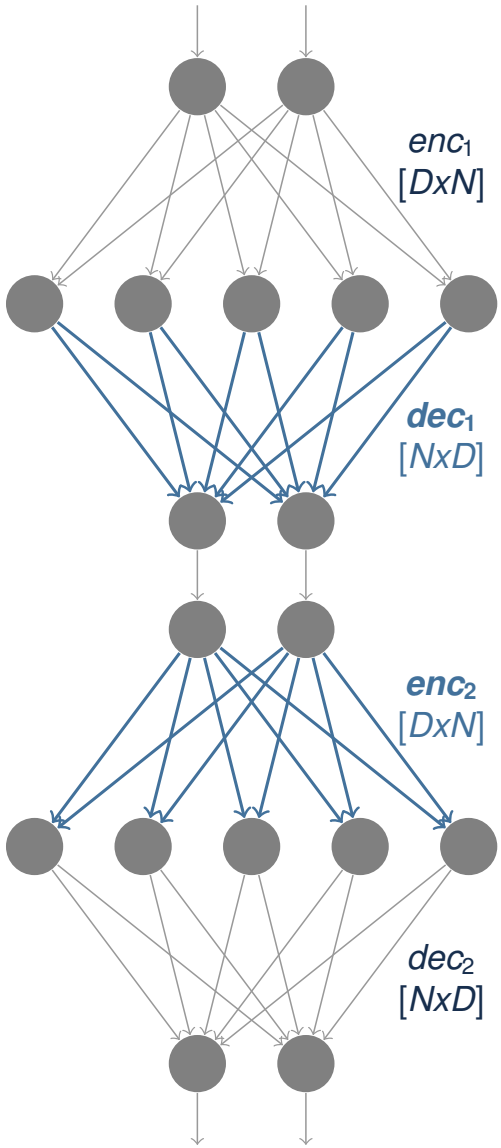




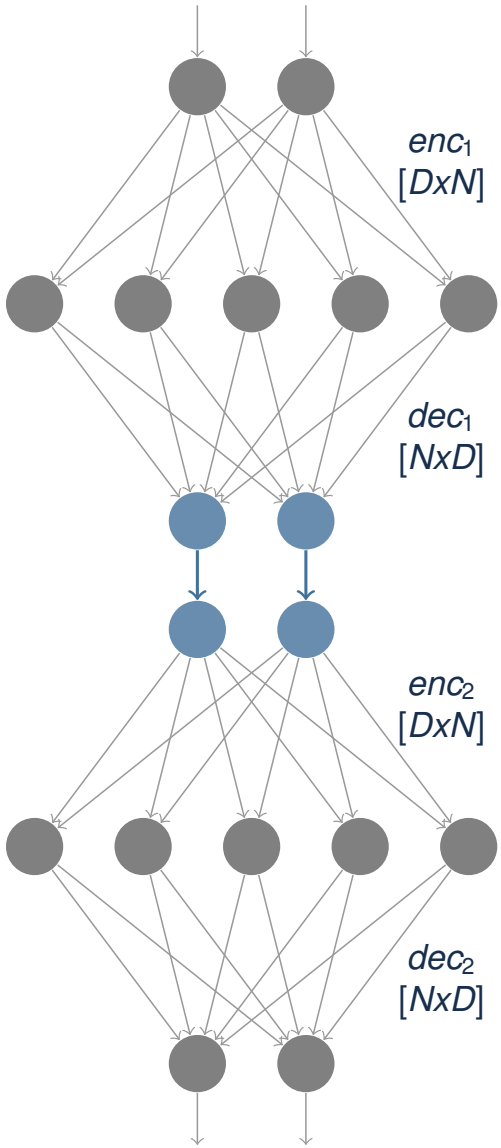
- We have $N \gg D$



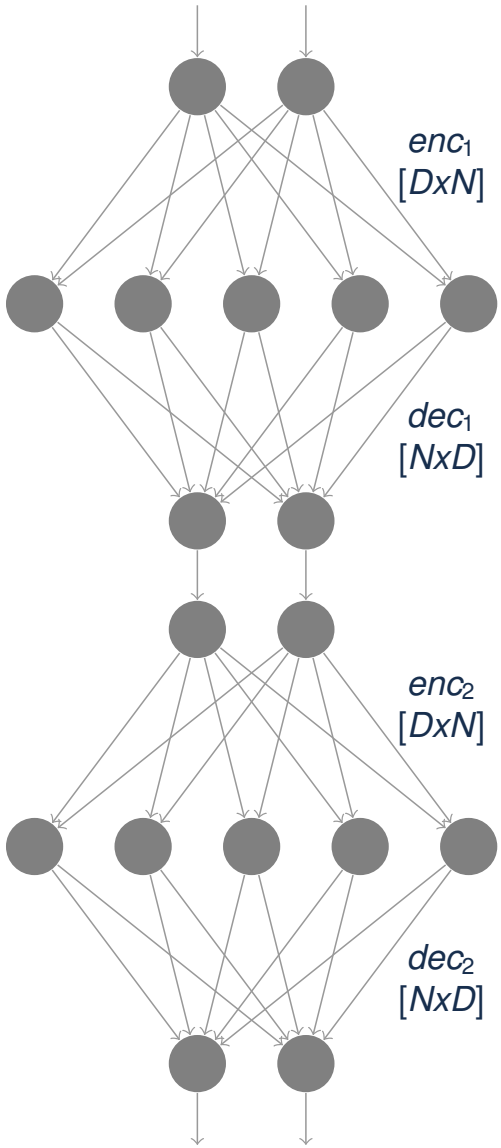
- We have $N \gg D$
  - ↪ NEF saves memory



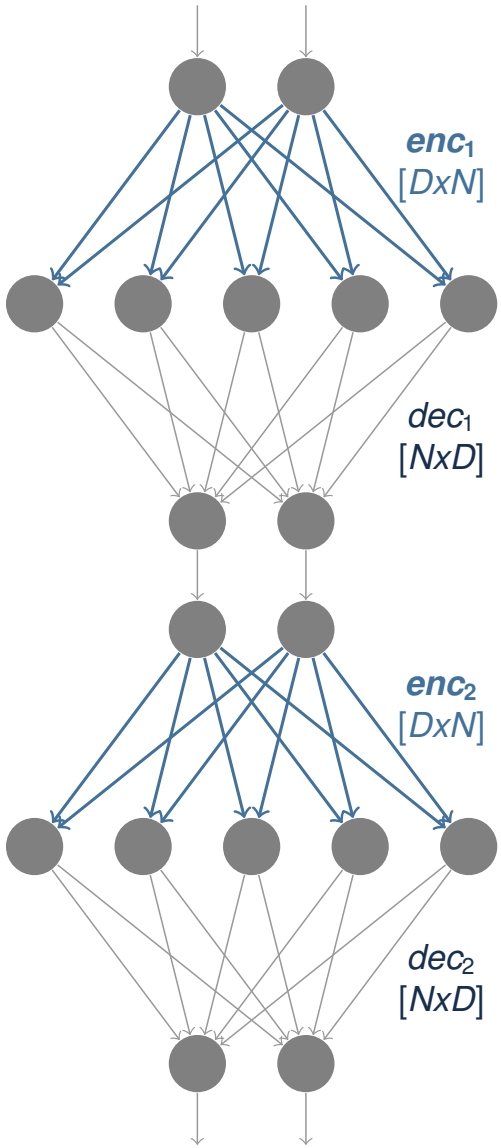
- We have $N \gg D$
  - ↪ NEF saves memory
  - ↪ NEF saves bandwidth



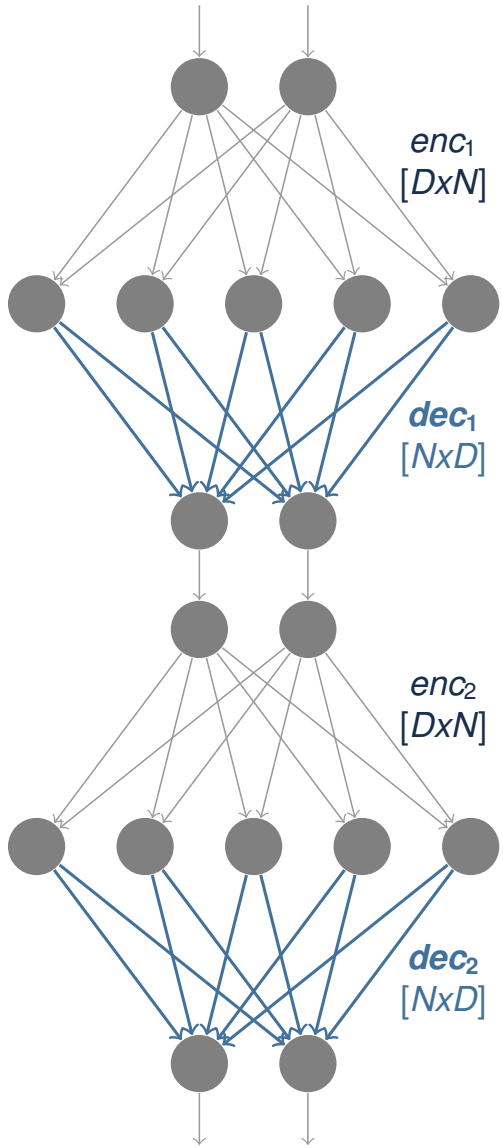
- We have $N \gg D$
  - ↪ NEF saves memory
  - ↪ NEF saves bandwidth
- NEF adds temporal dimension



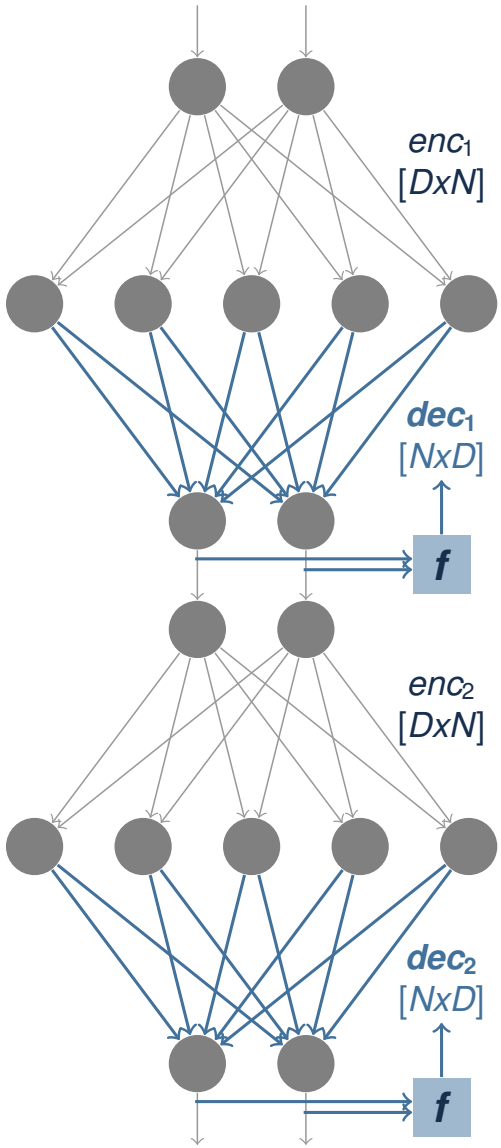
- Encoders are generated **randomly**



- Encoders are generated **randomly**
- Decoders can be:
  - ↳ Pre-solved in offline training and/or

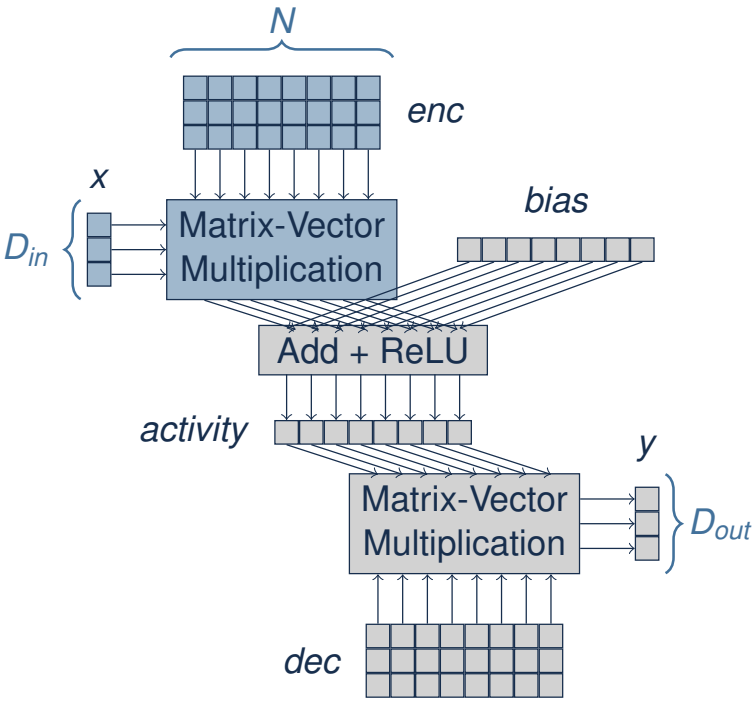


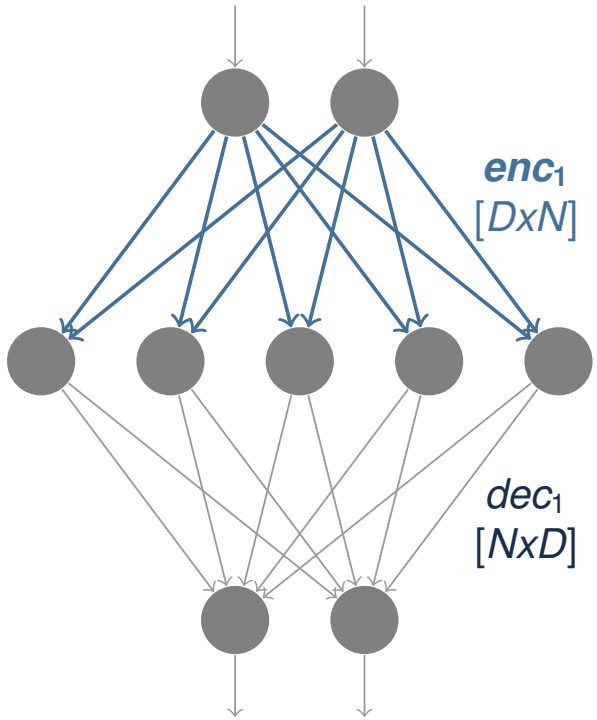
- Encoders are generated **randomly**
- Decoders can be:
  - ↳ Pre-solved in offline training and/or
  - ↳ Updated with online learning



This boils down to two matrix multiplies

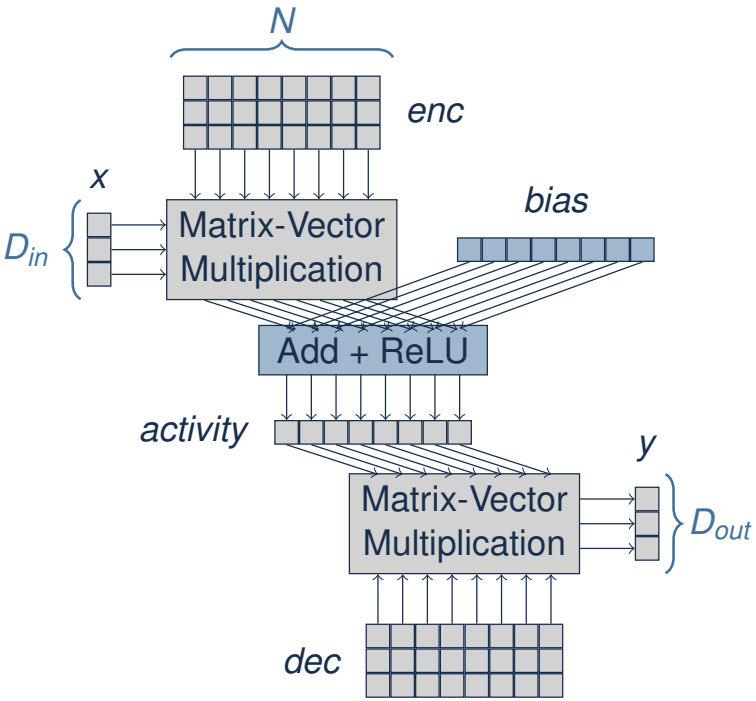
- One for encode

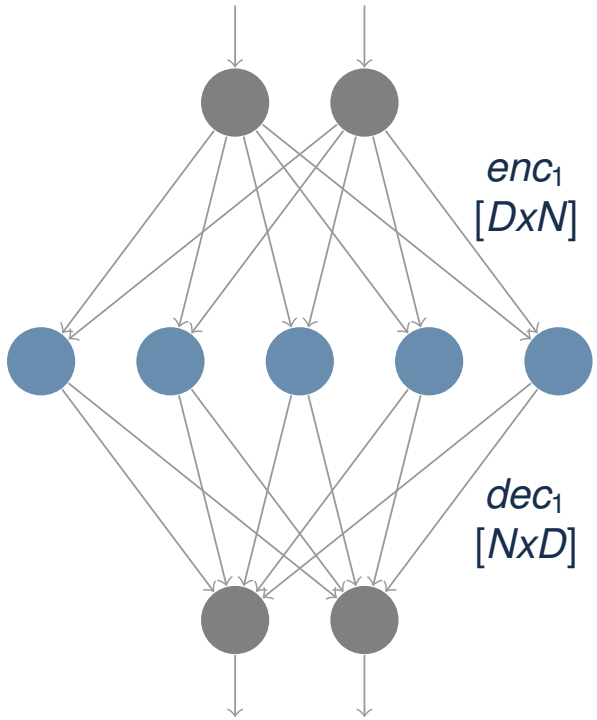




This boils down to two matrix multiplies

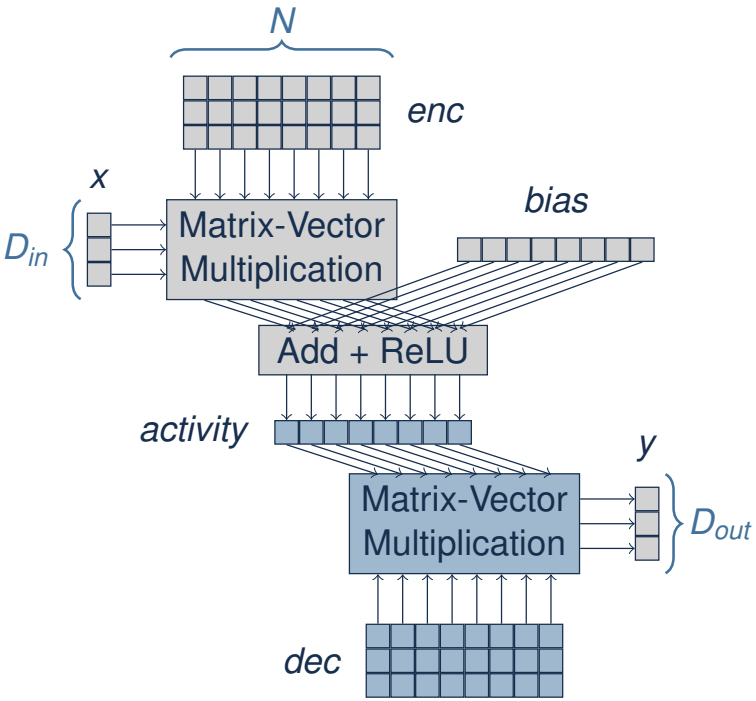
- One for encode
  - ↳ Plus neuron model (ReLU)

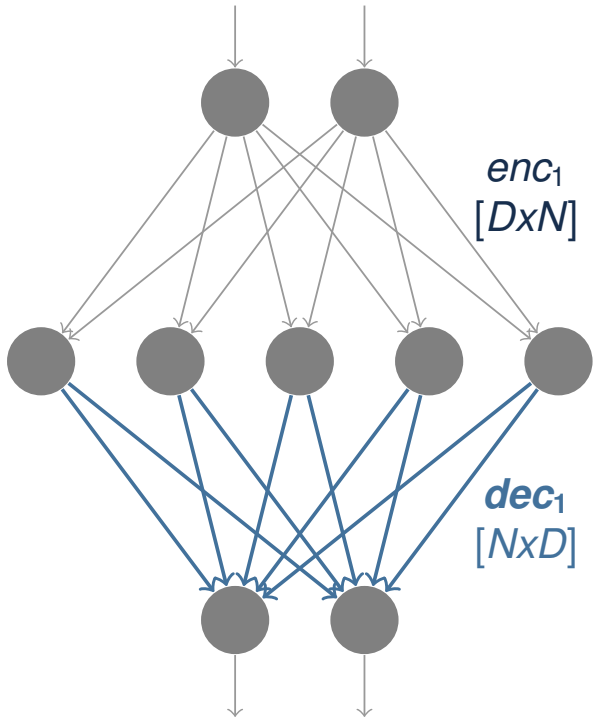




This boils down to two matrix multiplies

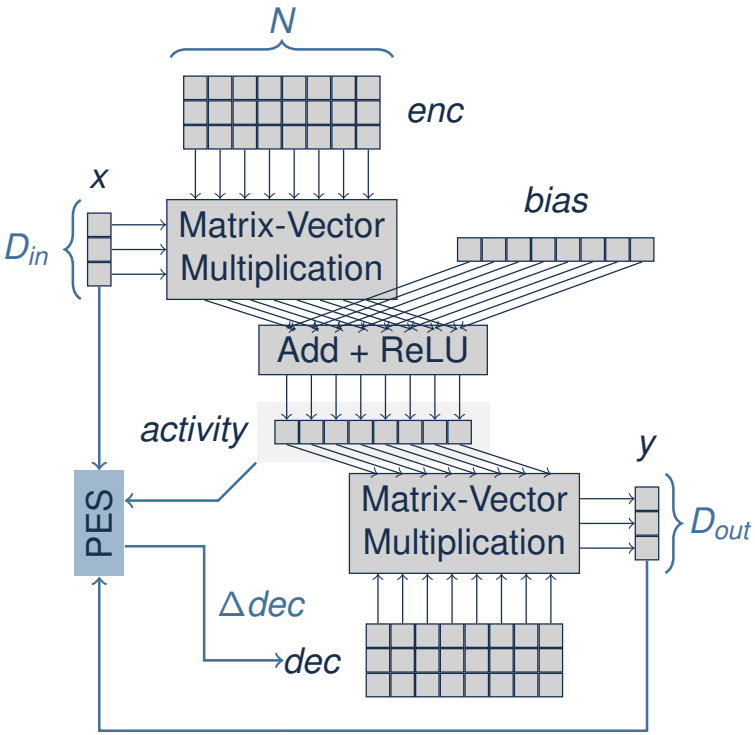
- One for encode
  - ↳ Plus neuron model (ReLU)
- One for decode

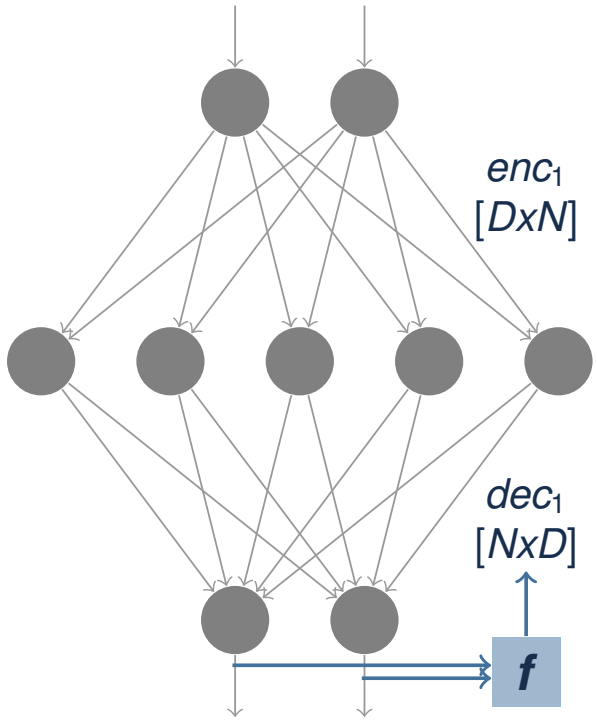




This boils down to two matrix multiplies

- One for encode
  - ↳ Plus neuron model (ReLU)
- One for decode
  - ↳ Plus online learning





Motivation

GPU is great for matrix multiplies!

- Jetson TX1 can do 1 TFLOP/s

Motivation

GPU is great for matrix multiplies!

- Jetson TX1 can do 1 TFLOP/s

However...

- No batching
- Limited on-chip memory
- No direct I/O access
- Little flexibility in data types

In Addition,

```
1 for i < N do
2 |   for j < D_in do
3 |   |   a_i += x_j * enc_ij
4 |   end
5 end
```

A single matrix multiply is simple to parallelize...

In Addition,

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```

A single matrix multiply is simple to parallelize...

...but two multiplies with different structure is less obvious.

The challenges we address are:

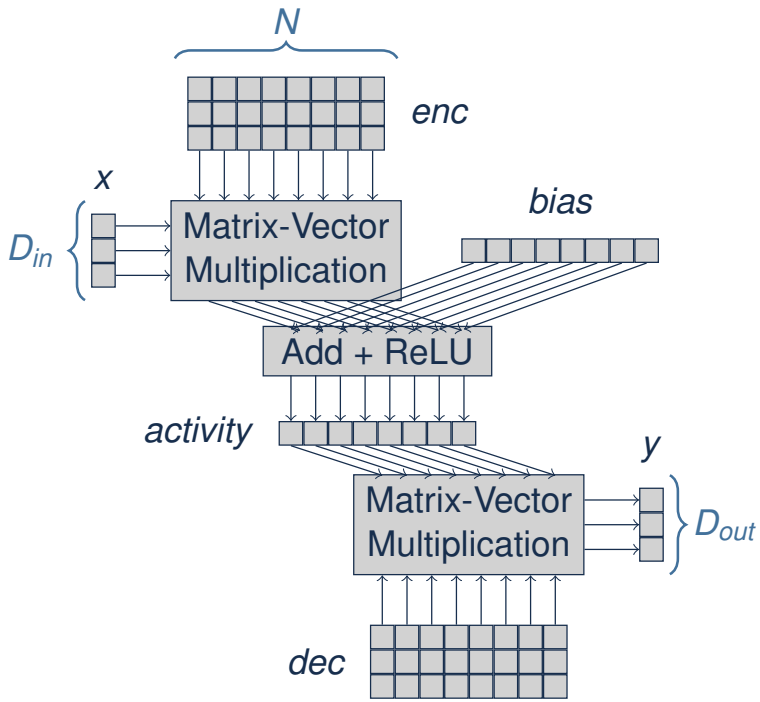
- Parallelization with HLS
  - ↳ Temporal dependency (learning) — **no batching!**
  - ↳ Structural difference between encode & decode
- Resource limitations on-chip

The challenges we address are:

- Parallelization with HLS
  - ↳ Temporal dependency (learning) — **no batching!**
  - ↳ Structural difference between encode & decode
- Resource limitations on-chip

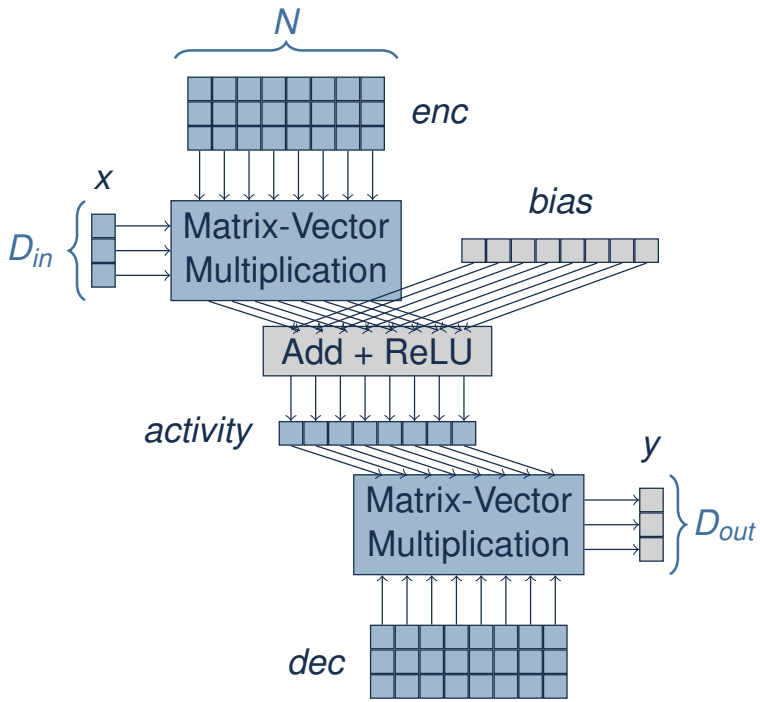
First let's estimate required cycles.

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



First let's estimate required cycles.

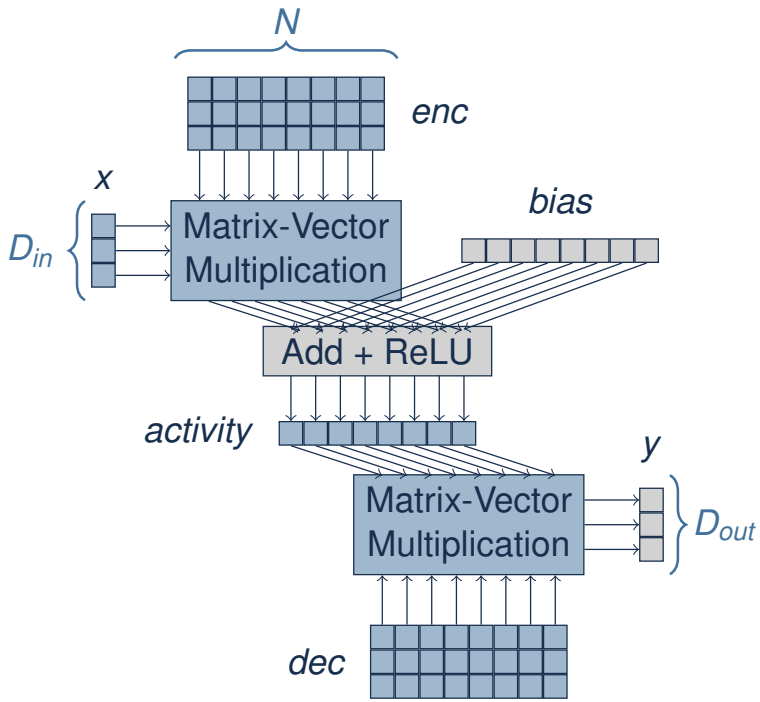
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



- $2 \times N \times D$

First let's estimate required cycles.

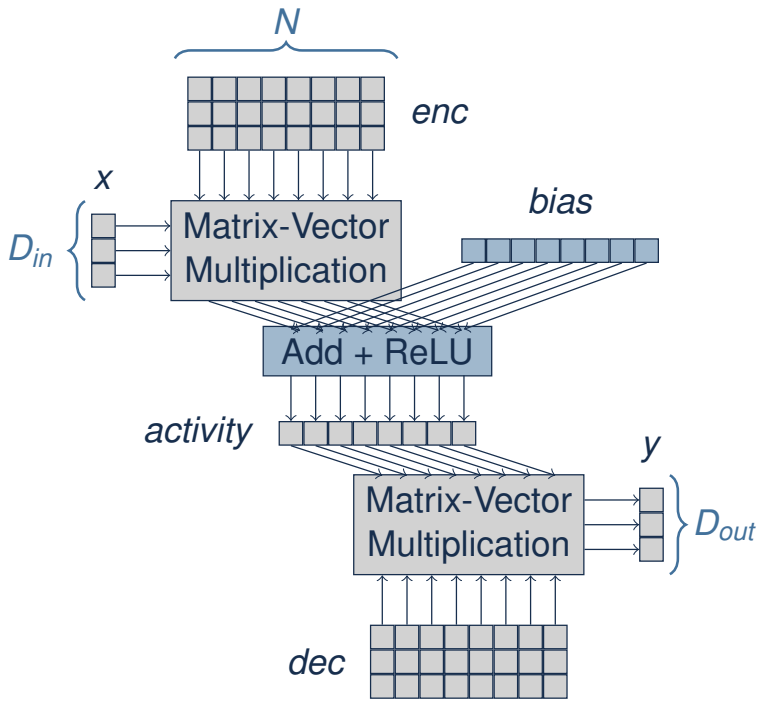
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



- $3 \times (2 \times N \times D)$

First let's estimate required cycles.

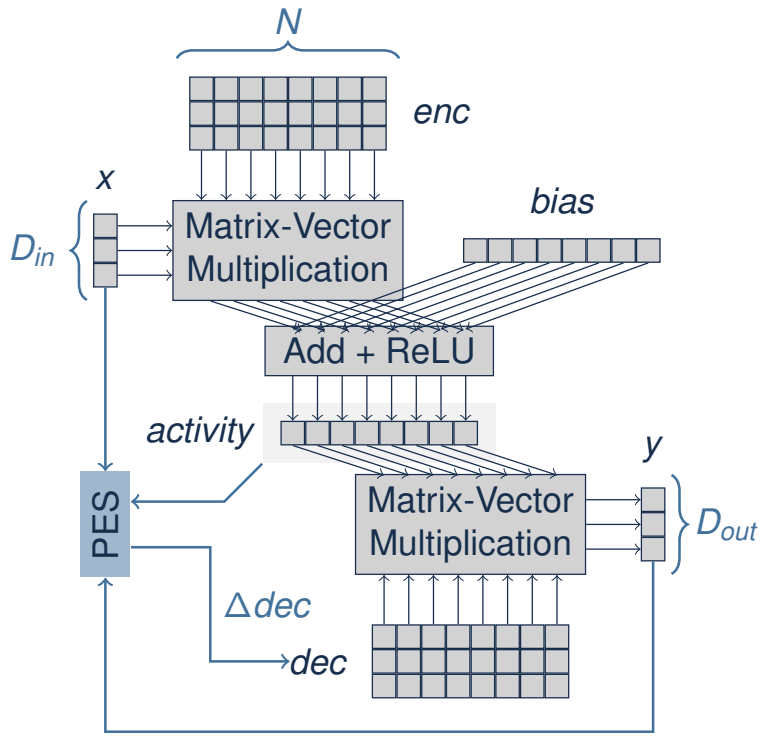
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



- $3 \times (2 \times N \times D) + 2 \times N$

First let's estimate required cycles.

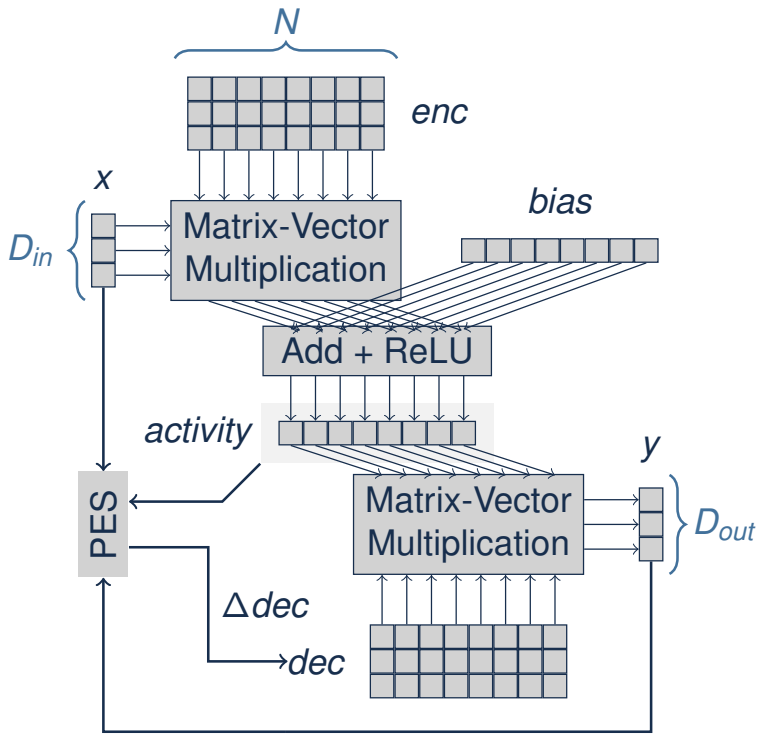
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



- $3 \times (3 \times N \times D) + 2 \times N$

First let's estimate required cycles.

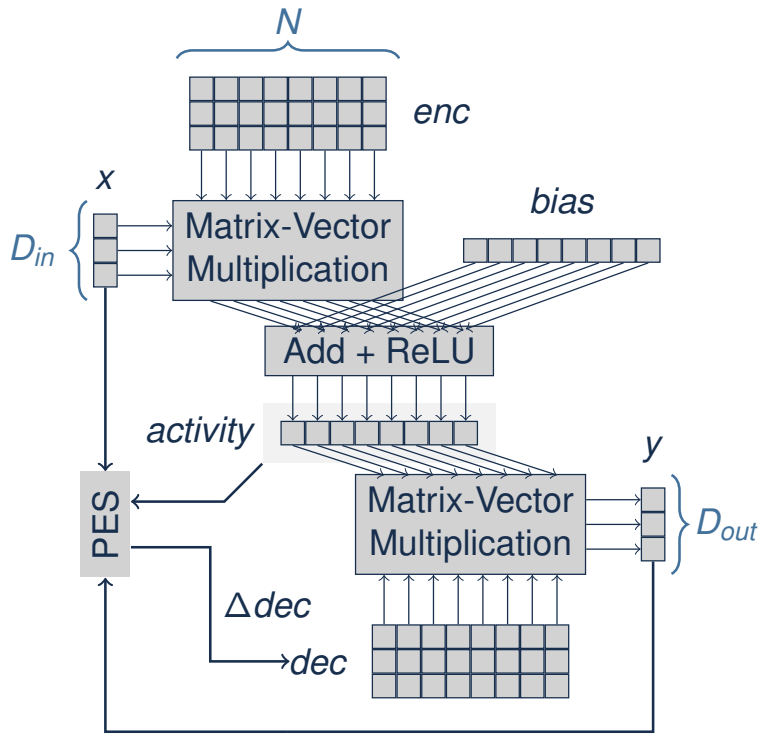
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



- $3 \times (3 \times N \times D) + 2 \times N$
- *e.g.* N=200 and D=2
  - ↪ = 4000 cycles

First let's estimate required cycles.

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```

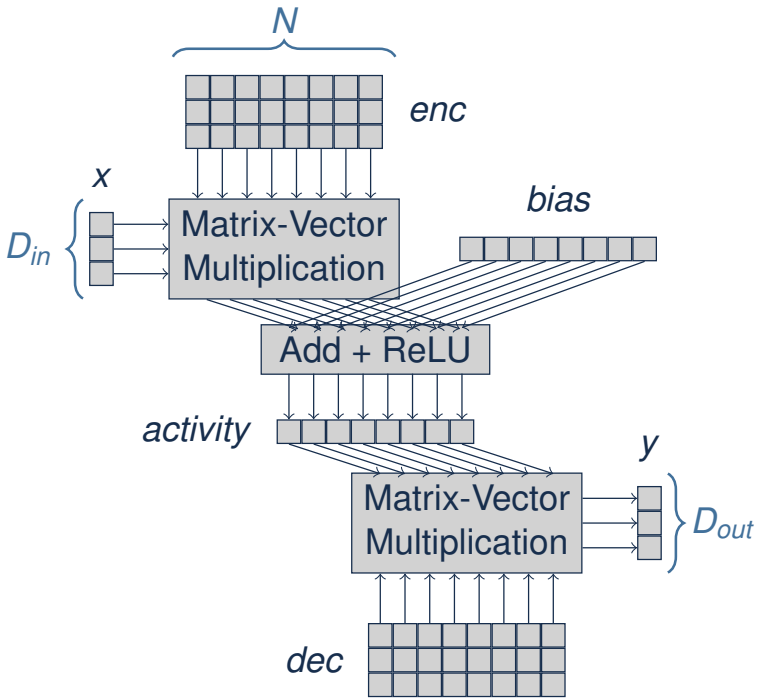


- $3 \times (3 \times N \times D) + 2 \times N$
- *e.g.* N=200 and D=2
  - ↪ = 4000 cycles

but HLS estimates 6000 cycles with naive approach!

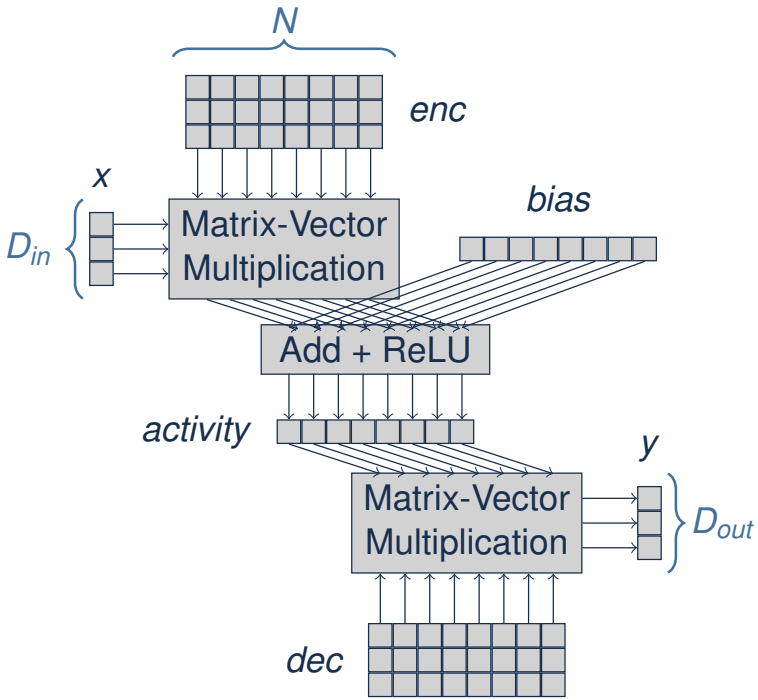
Now we restructure the second matrix multiply.

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for j < D_out do
7  |   for i < N do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



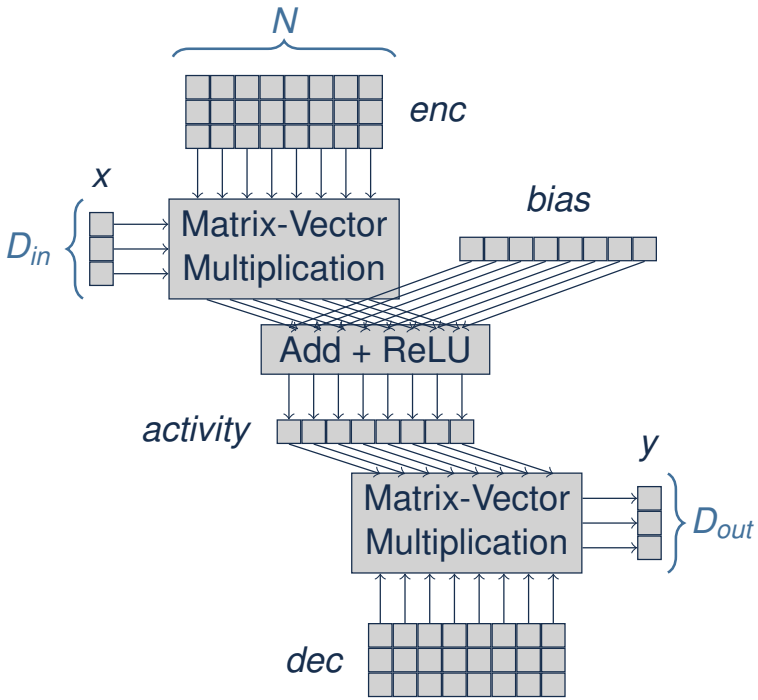
Now we restructure the second matrix multiply.

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for i < N do
7  |   for j < D_out do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



Now we restructure the second matrix multiply.

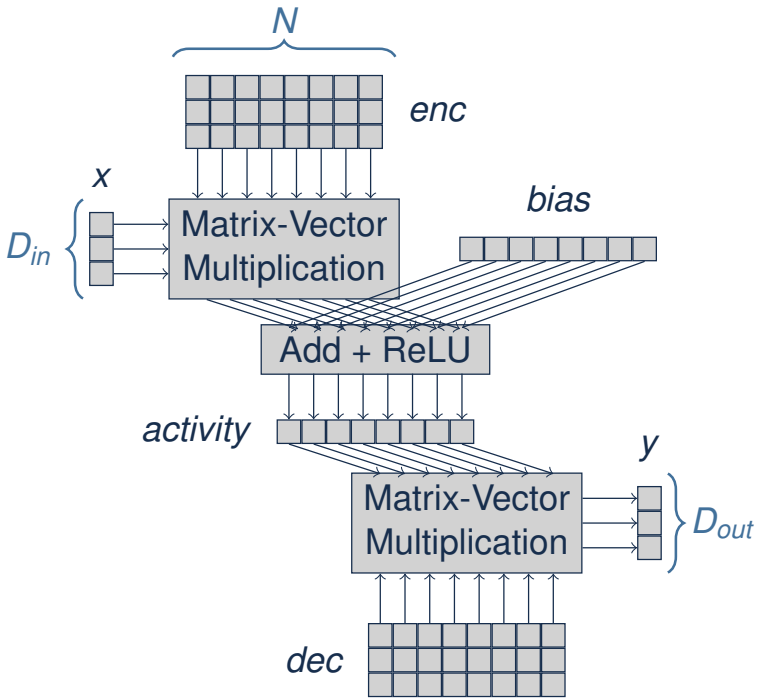
```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for i < N do
7  |   for j < D_out do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



This improved `II` and got us to 4800 cycles.
(recall we estimated 4000)

Now we restructure the second matrix multiply.

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for i < N do
7  |   for j < D_out do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



This improved `II` and got us to 4800 cycles.
(recall we estimated 4000)

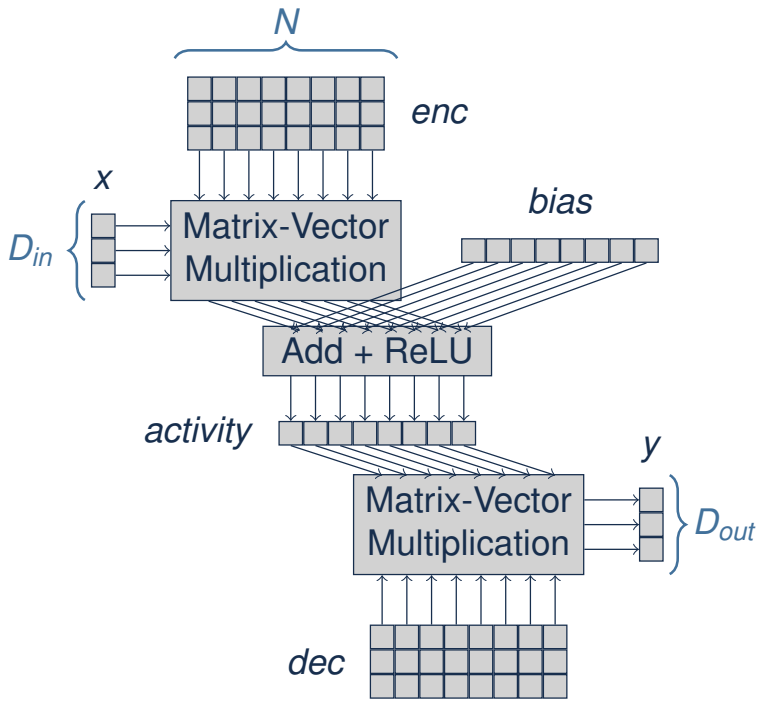
Now loop structure is harmonized, we want to parallelize

Now loop structure is harmonized, we want to parallelize

But `unroll` and `dataflow` pragmas could not automatically extract parallelism

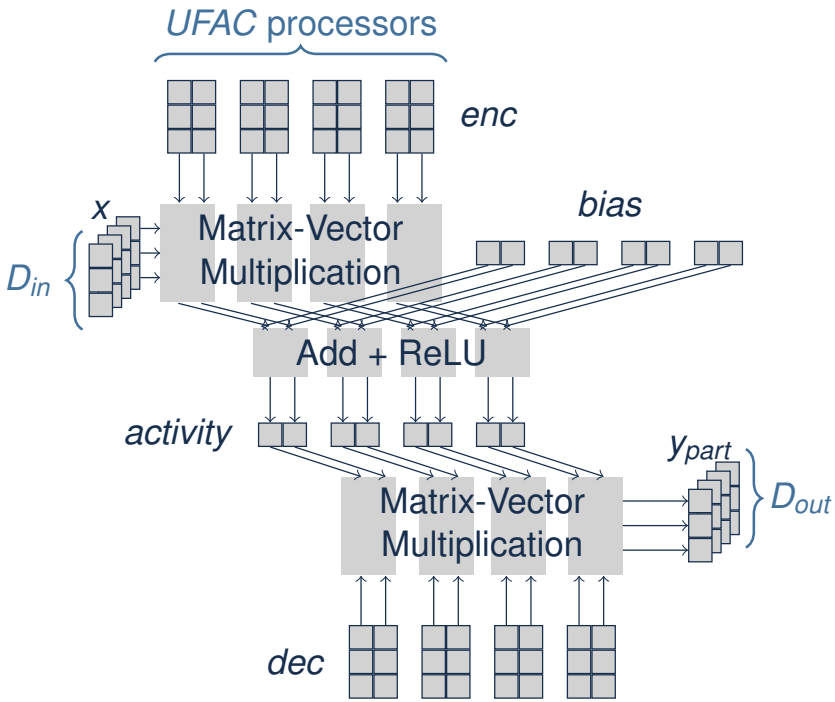
We add explicit parallel structure

```
1  for i < N do
2  |   for j < D_in do
3  |   |   a_i += x_j * enc_ij
4  |   end
5  end
6  for i < N do
7  |   for j < D_out do
8  |   |   y_j += a_i * dec_ji
9  |   end
10 end
```



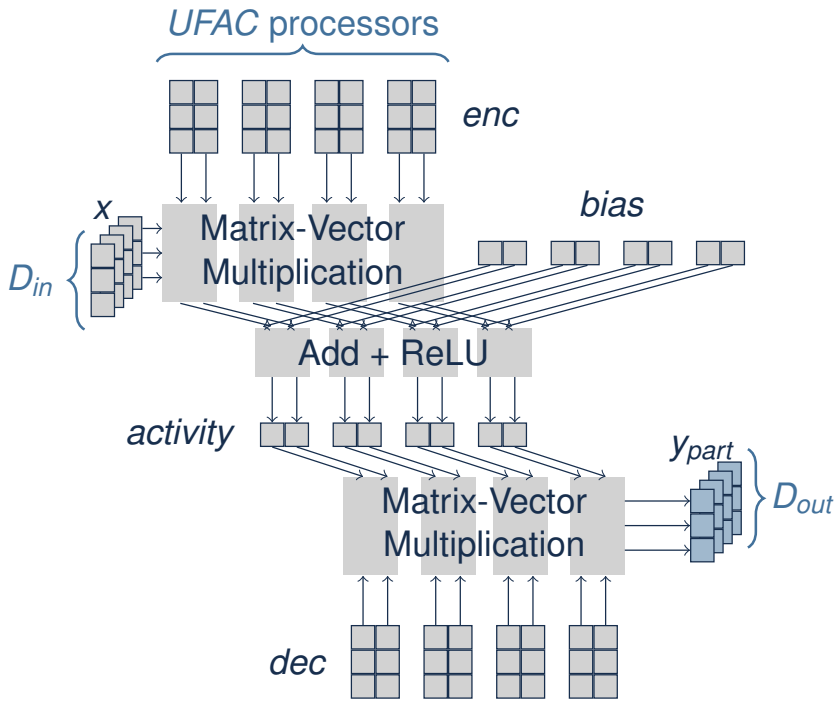
We add explicit parallel structure

```
1  for k < UFAC do
2      for i < ⌈N / UFAC⌉ do
3          for j < D_in do
4              a_i += x_kj * enc_ij
5          end
6      end
7      for i < ⌈N / UFAC⌉ do
8          for j < D_out do
9              y^part_kj += a_i * dec_ji
10         end
11     end
12 end
```



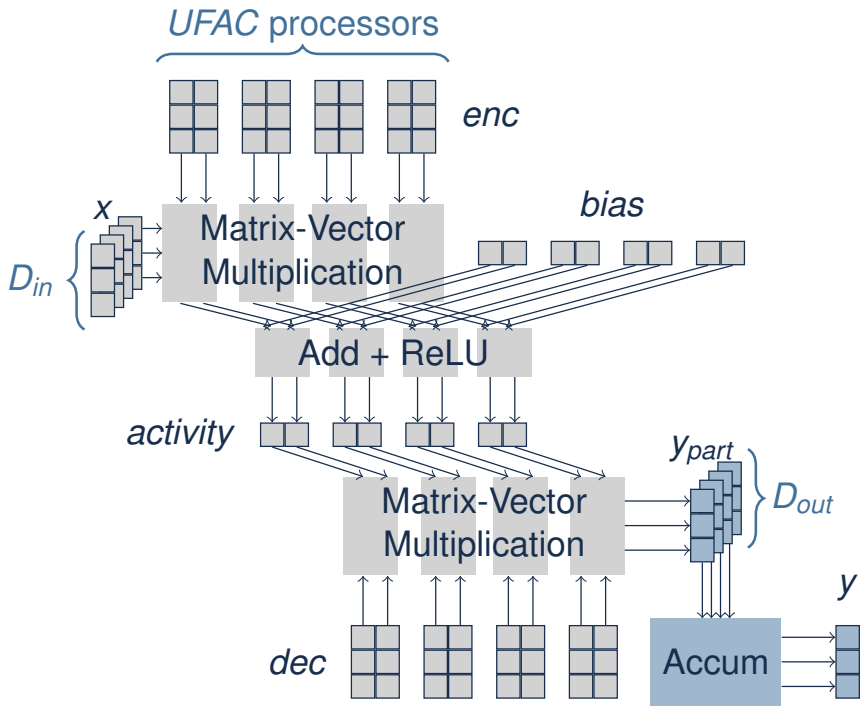
Which creates partial results that must be accumulated

```
1  for k < UFAC do
2      for i < ⌈N / UFAC⌉ do
3          for j < D_in do
4              a_i += x_kj * enc_ij
5          end
6      end
7      for i < ⌈N / UFAC⌉ do
8          for j < D_out do
9              y_kj^part += a_i * dec_ji
10         end
11     end
12 end
```



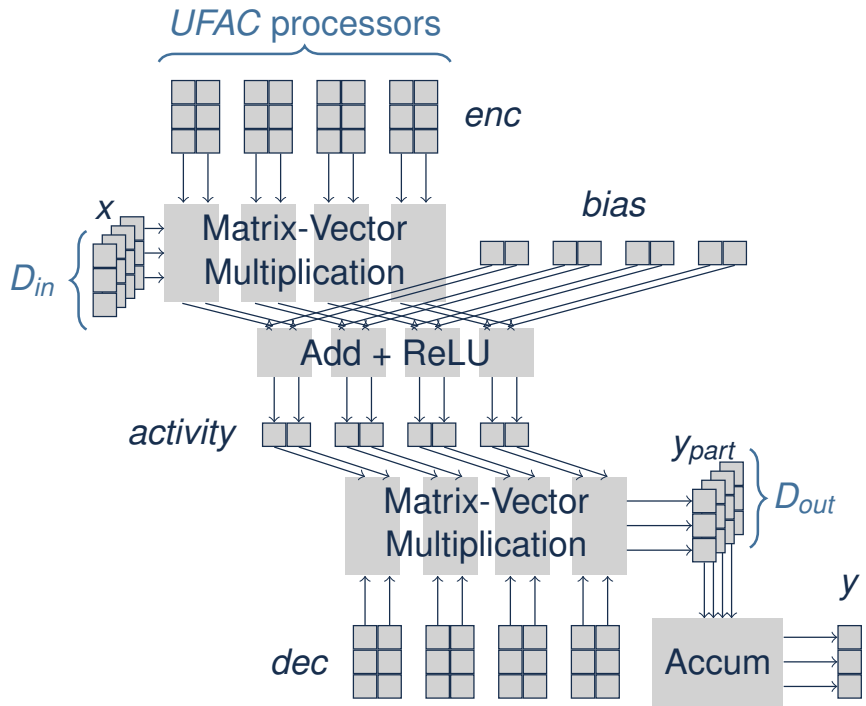
Which creates partial results that must be accumulated

```
1  for k < UFAC do
2      for i < ⌈N / UFAC⌉ do
3          for j < D_in do
4              a_i += x_kj * enc_ij
5          end
6      end
7      for i < ⌈N / UFAC⌉ do
8          for j < D_out do
9              y_kj^part += a_i * dec_ji
10         end
11     end
12 end
13 for j < D_out do
14     for k < UFAC do
15         y_j += y_kj^part
16     end
17 end
```



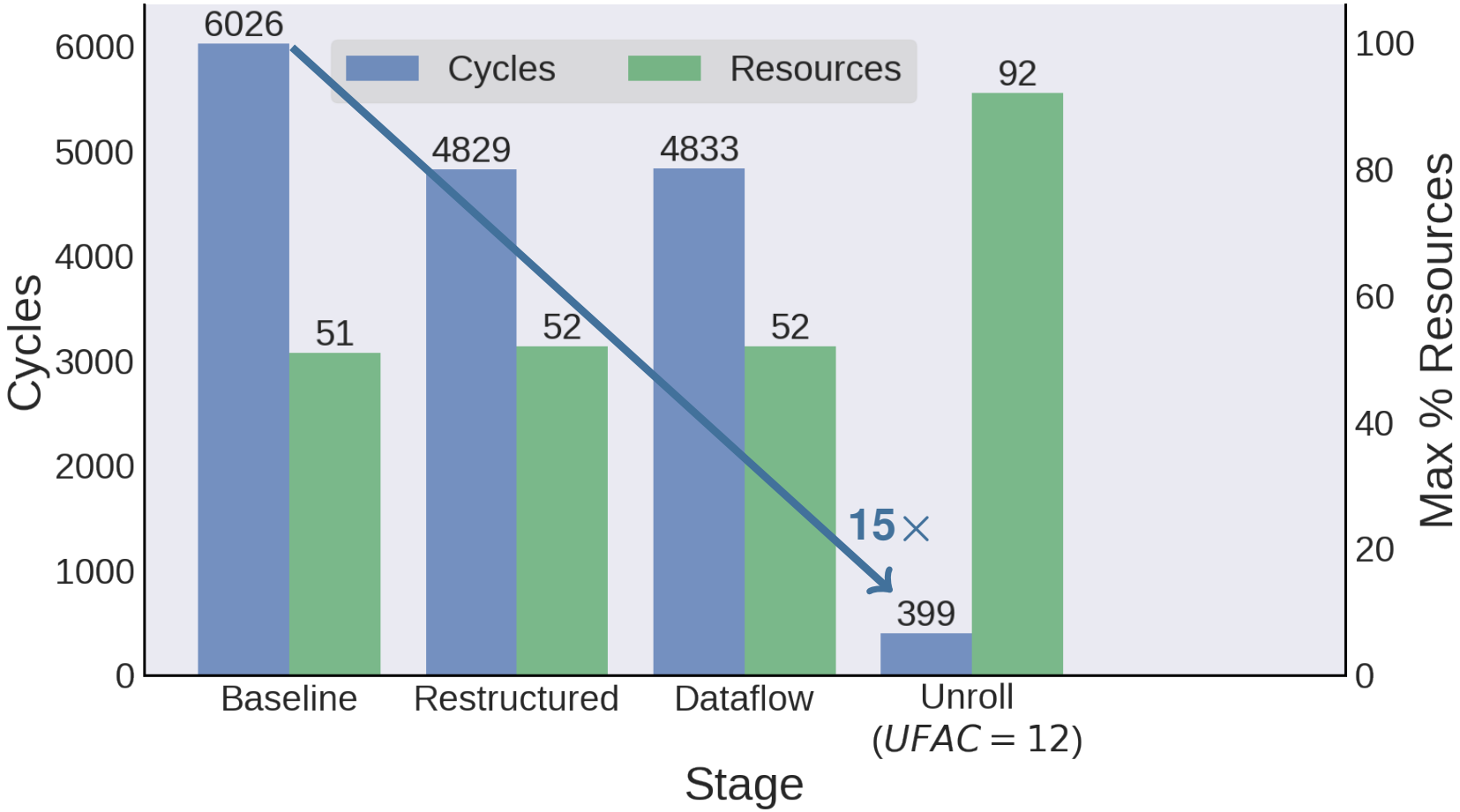
Which creates partial results that must be accumulated

```
1  for k < UFAC do
2      for i < ⌈N / UFAC⌉ do
3          for j < D_in do
4              a_i += x_kj * enc_ij
5          end
6      end
7      for i < ⌈N / UFAC⌉ do
8          for j < D_out do
9              y^part_kj += a_i * dec_ji
10         end
11     end
12 end
13 for j < D_out do
14     for k < UFAC do
15         y_j += y^part_kj
16     end
17 end
```

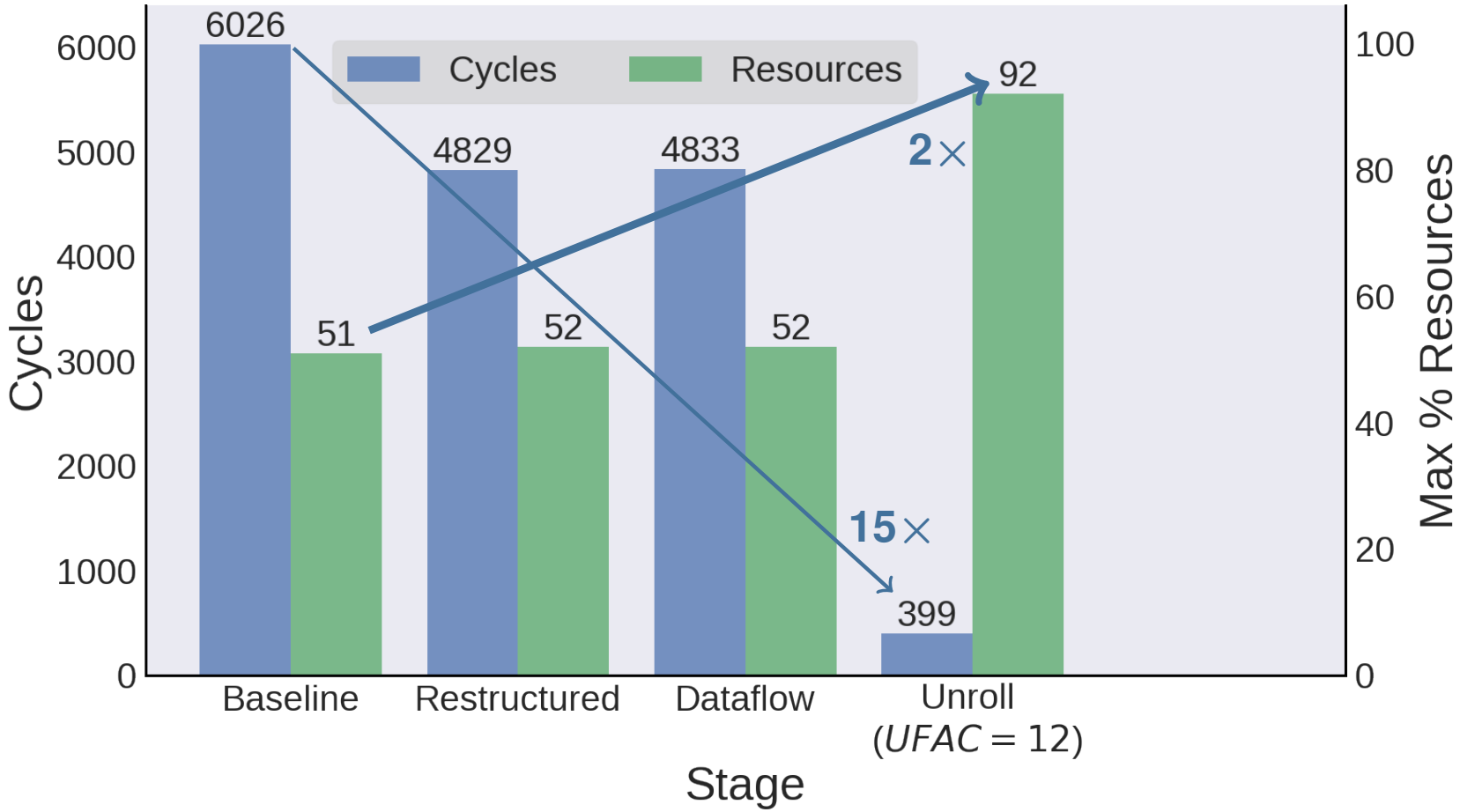


400 cycles with $UFAC = 12$.

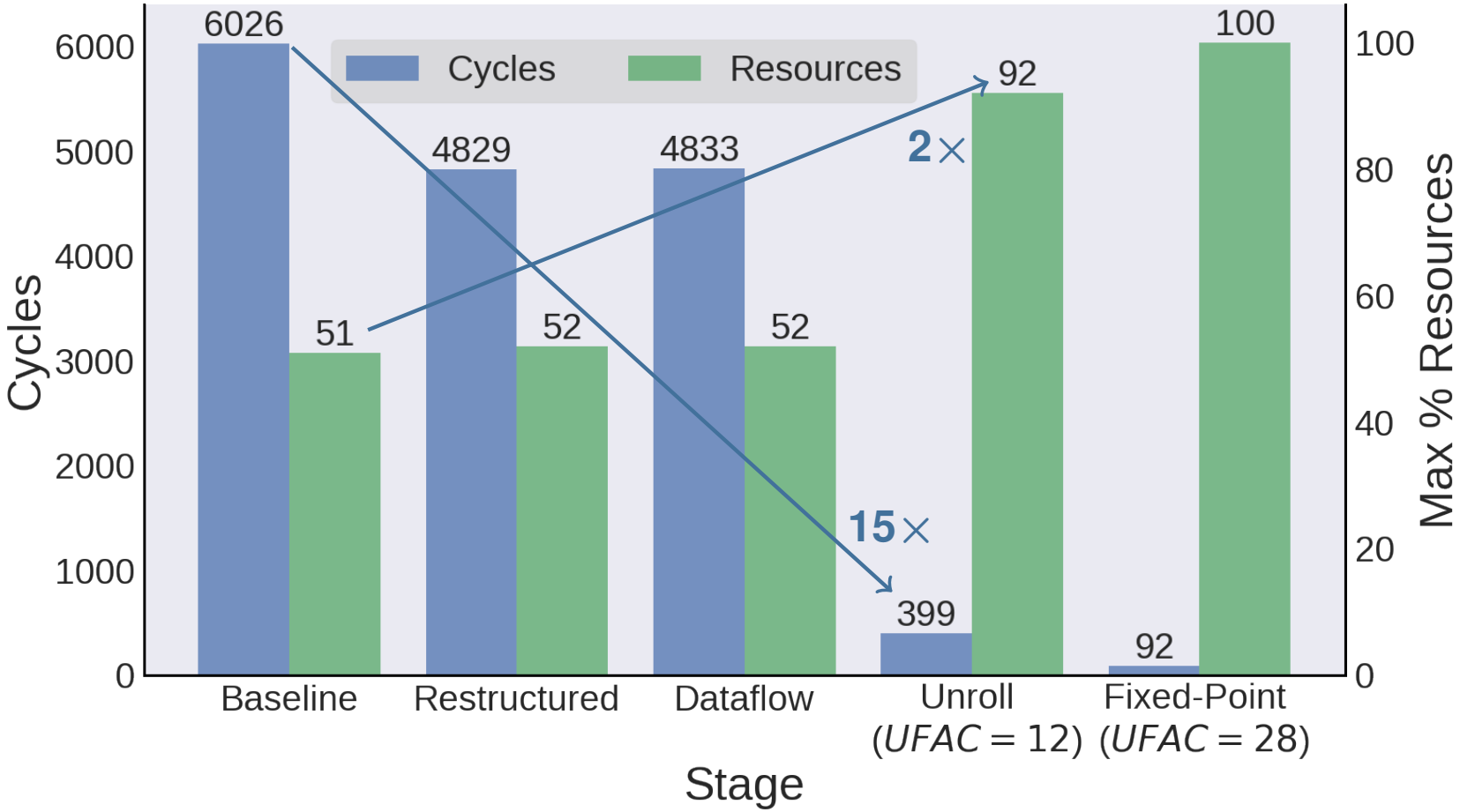
Clever HLS structure allows us to use the entire chip



Clever HLS structure allows us to use the entire chip



Fixed-point further shrinks compute tile



The challenges we address are:

- Parallelization with HLS
  - ↳ Temporal dependency (learning) — **no batching!**
  - ↳ Structural difference between encode & decode
- Resource limitations on-chip

We create 5 different fixed-point types using the `ap_fixed` template type provided by Vivado HLS

We create 5 different fixed-point types using the `ap_fixed` template type provided by Vivado HLS

For each type we have many choices for:

- Number of word bits
- Number of integer bits
- Rounding strategy
- Overflow strategy

We create 5 different fixed-point types using the `ap_fixed` template type provided by Vivado HLS

For each type we have many choices for:

- Number of word bits
- Number of integer bits
- Rounding strategy
- Overflow strategy

This results in $\approx 10^{20}$ configurations!

We use a Python package called Hyperopt to automatically tune our hyper-parameters and extract Pareto optimal designs for a given cost function.

We want to minimize error and maximize performance

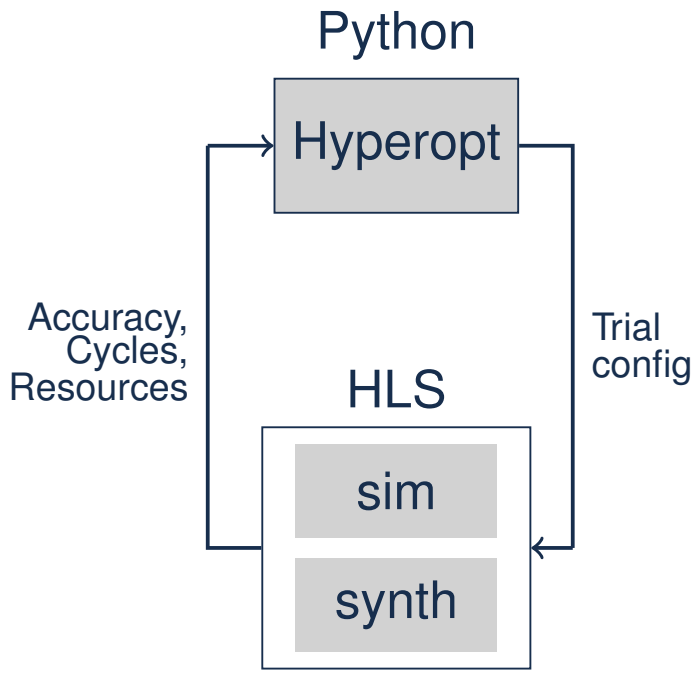
We want to minimize error and maximize performance

We use Hyperopt to minimize error and resources

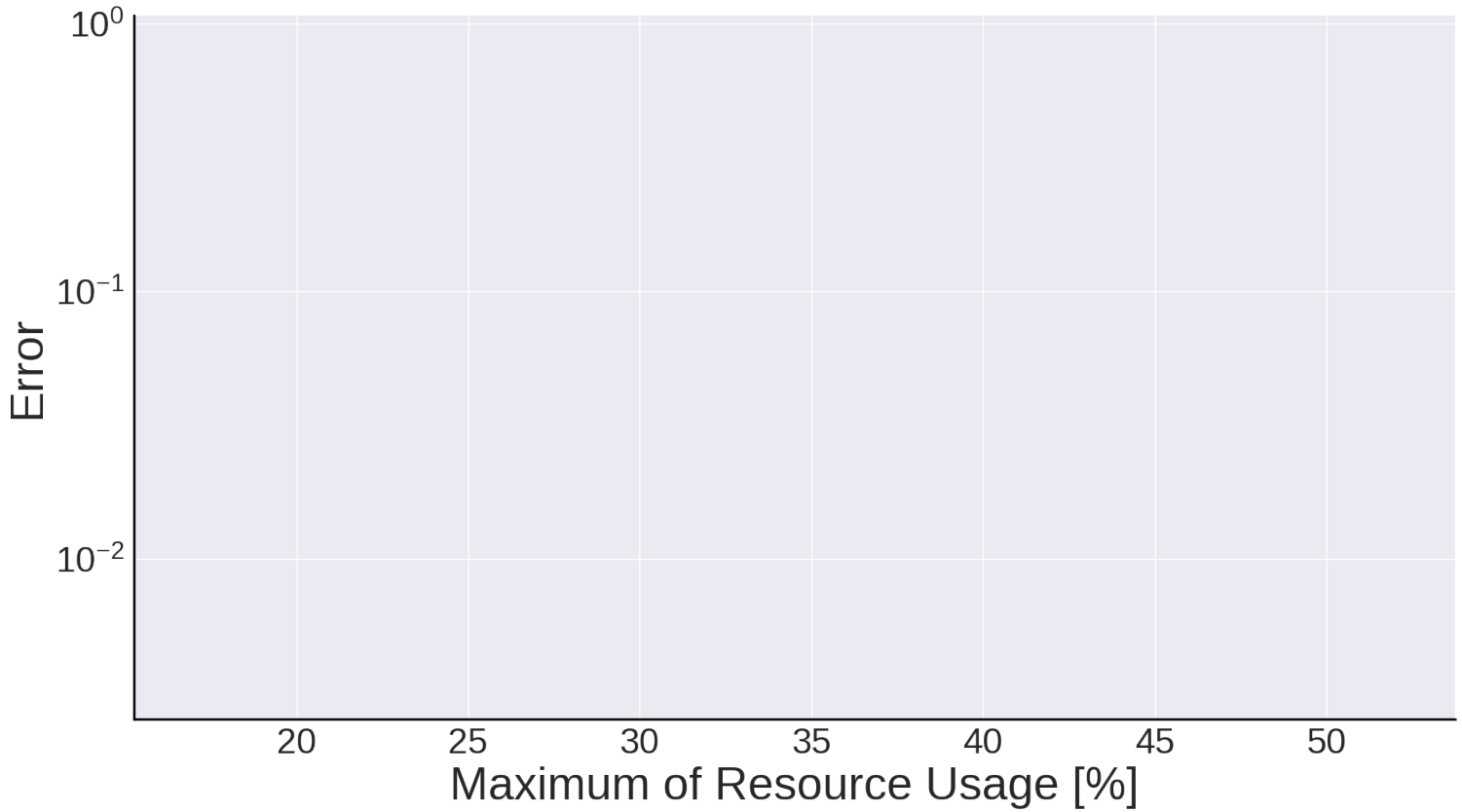
- Small, high-performance tile

Each Hyperopt trial runs:

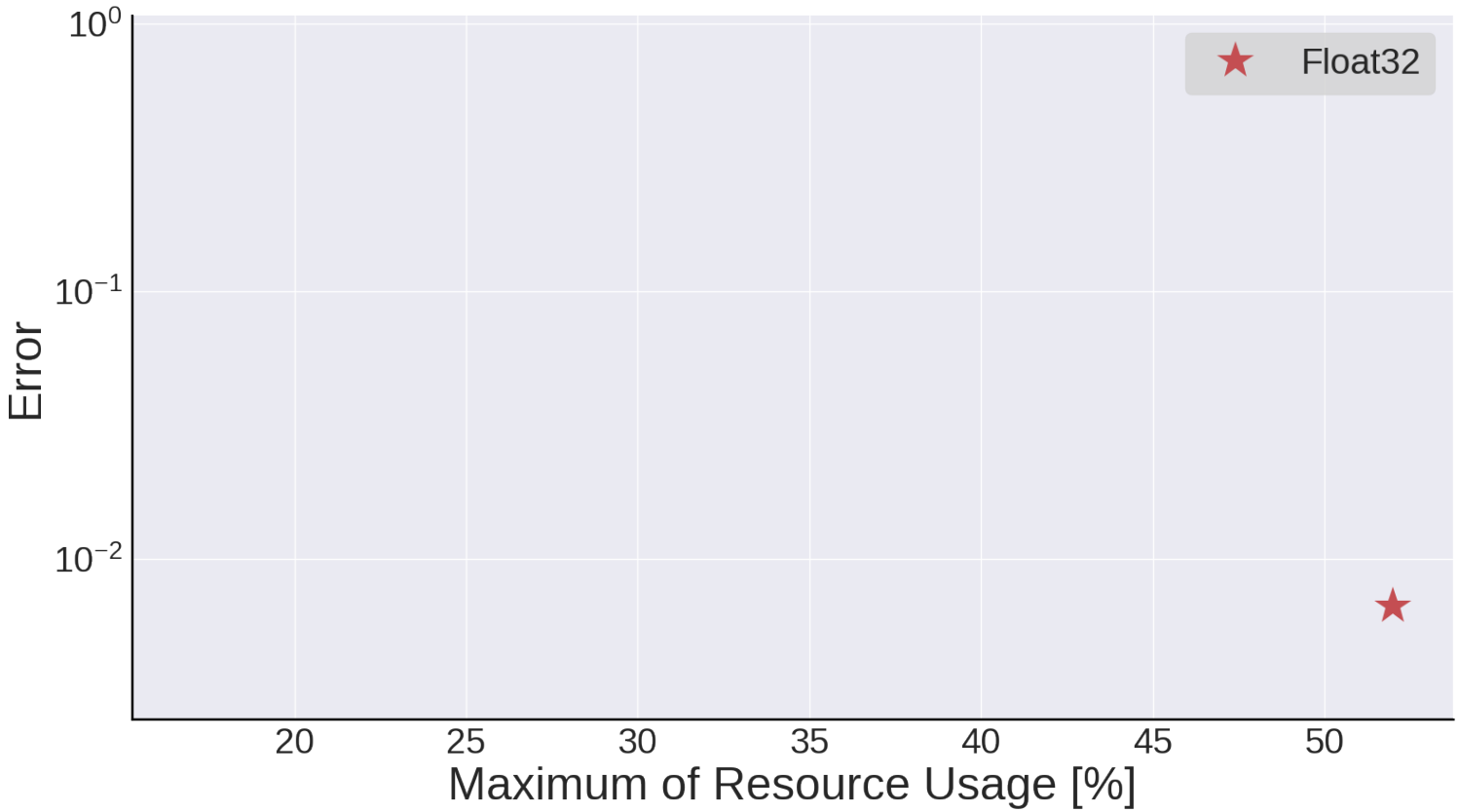
- C simulation
- C synthesis (no P & R)



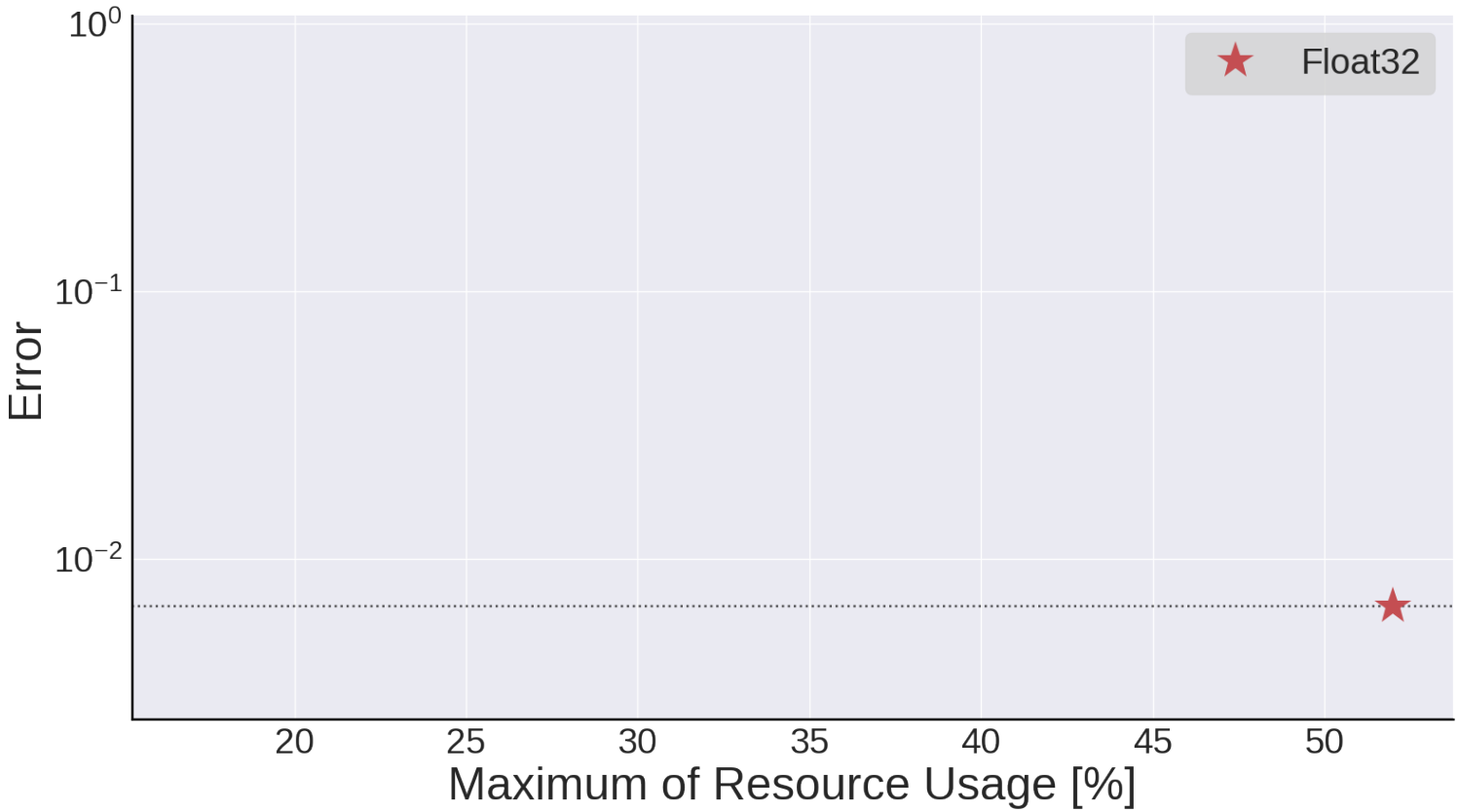
Let's look at some results:



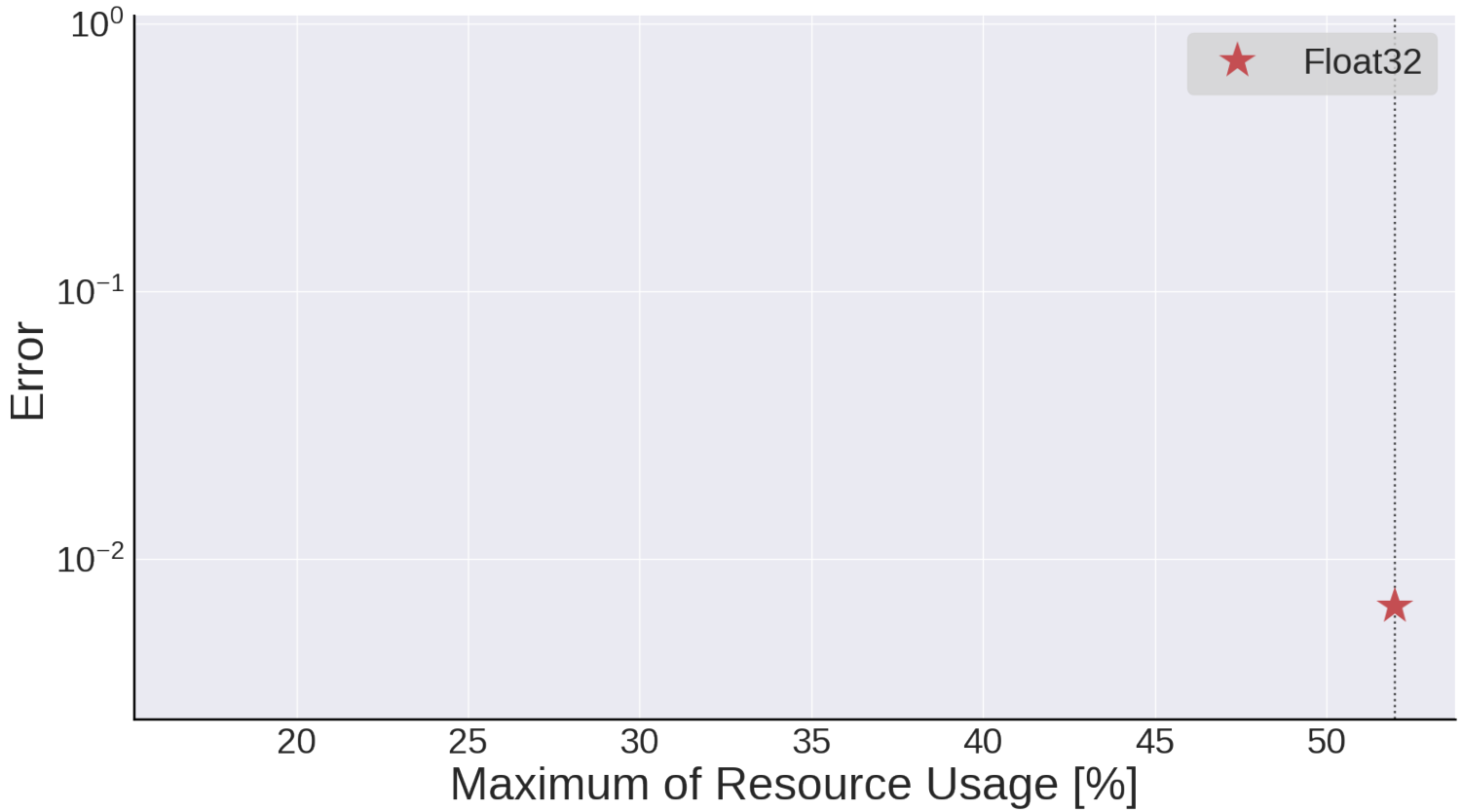
Let's look at some results:



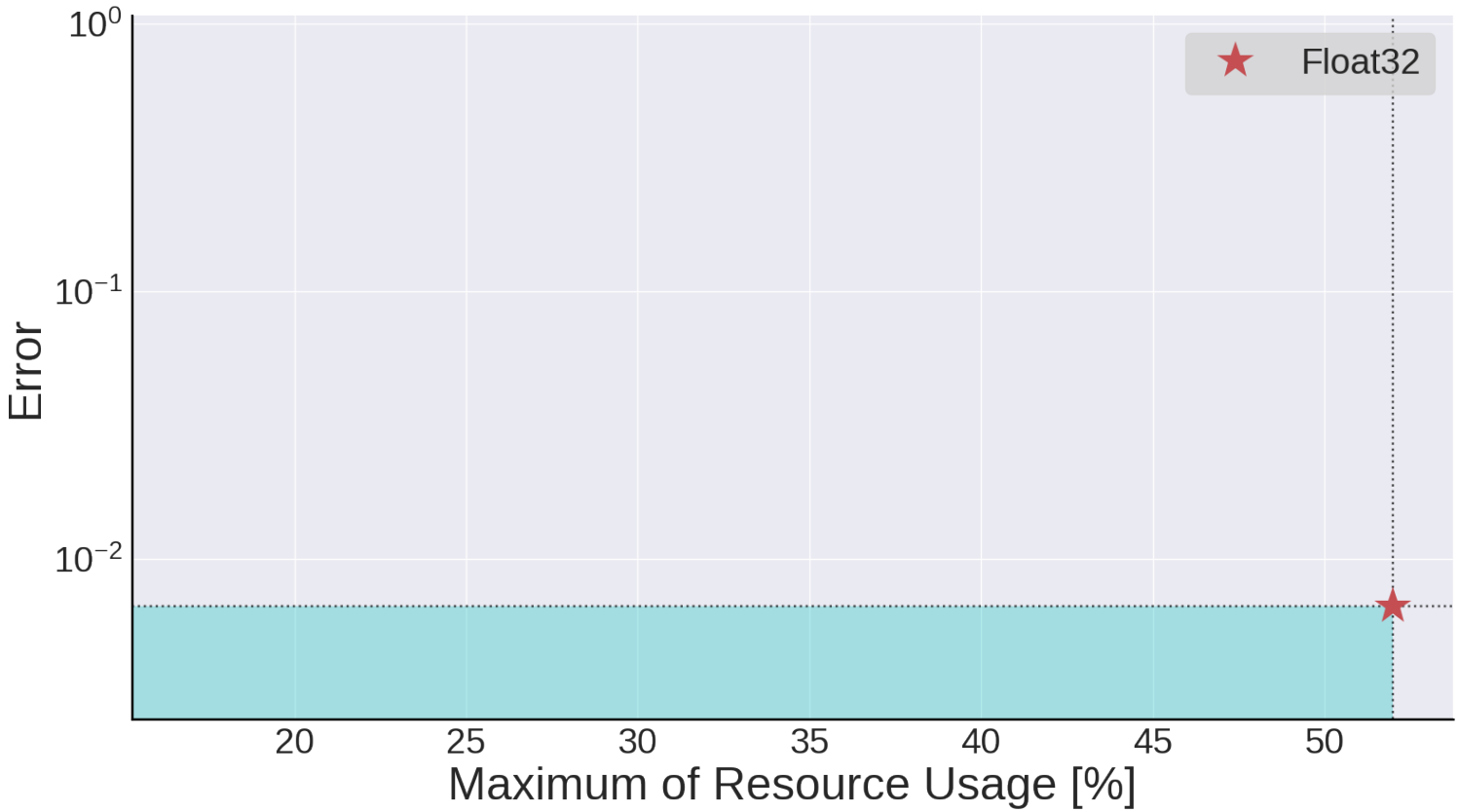
Let's look at some results:



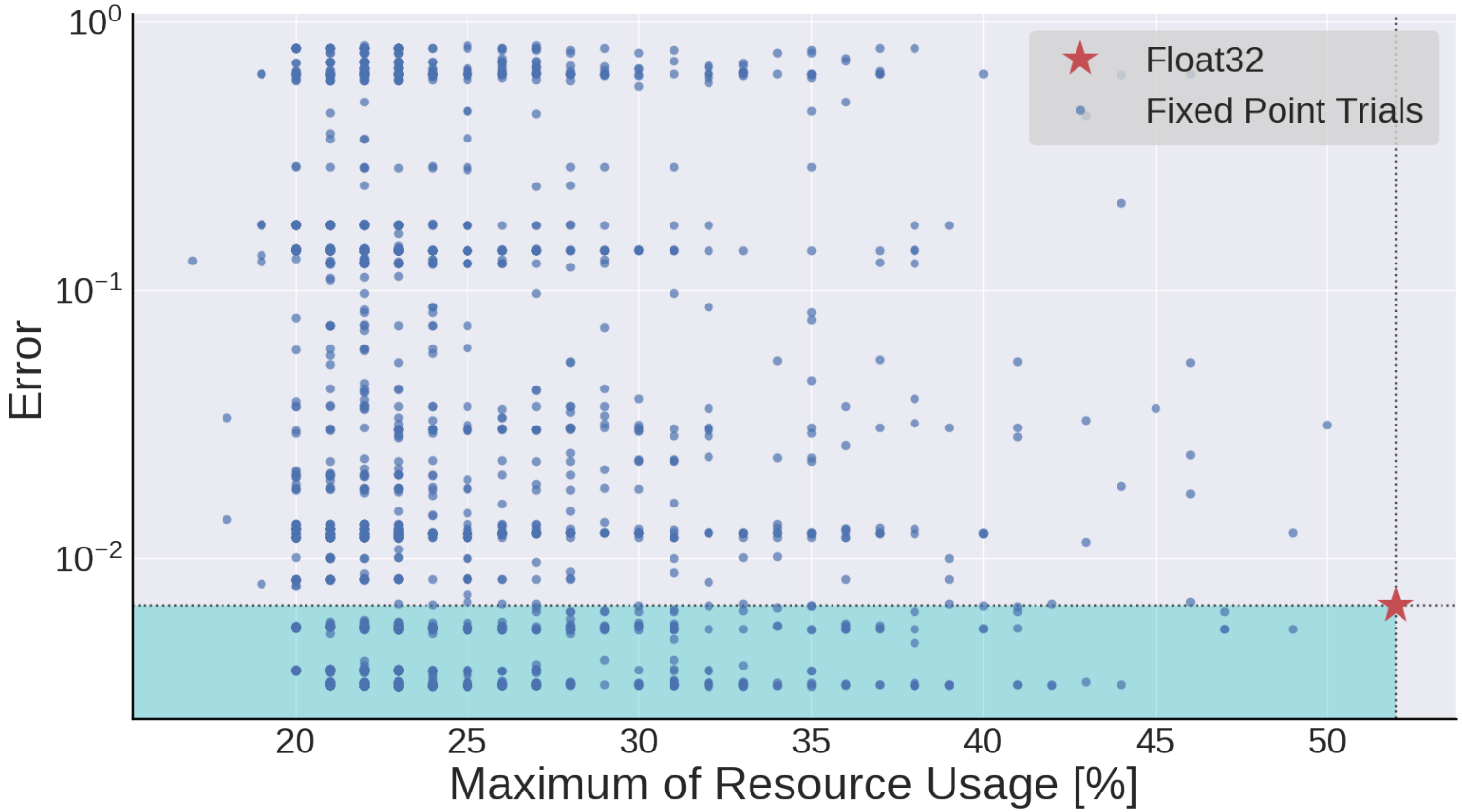
Let's look at some results:



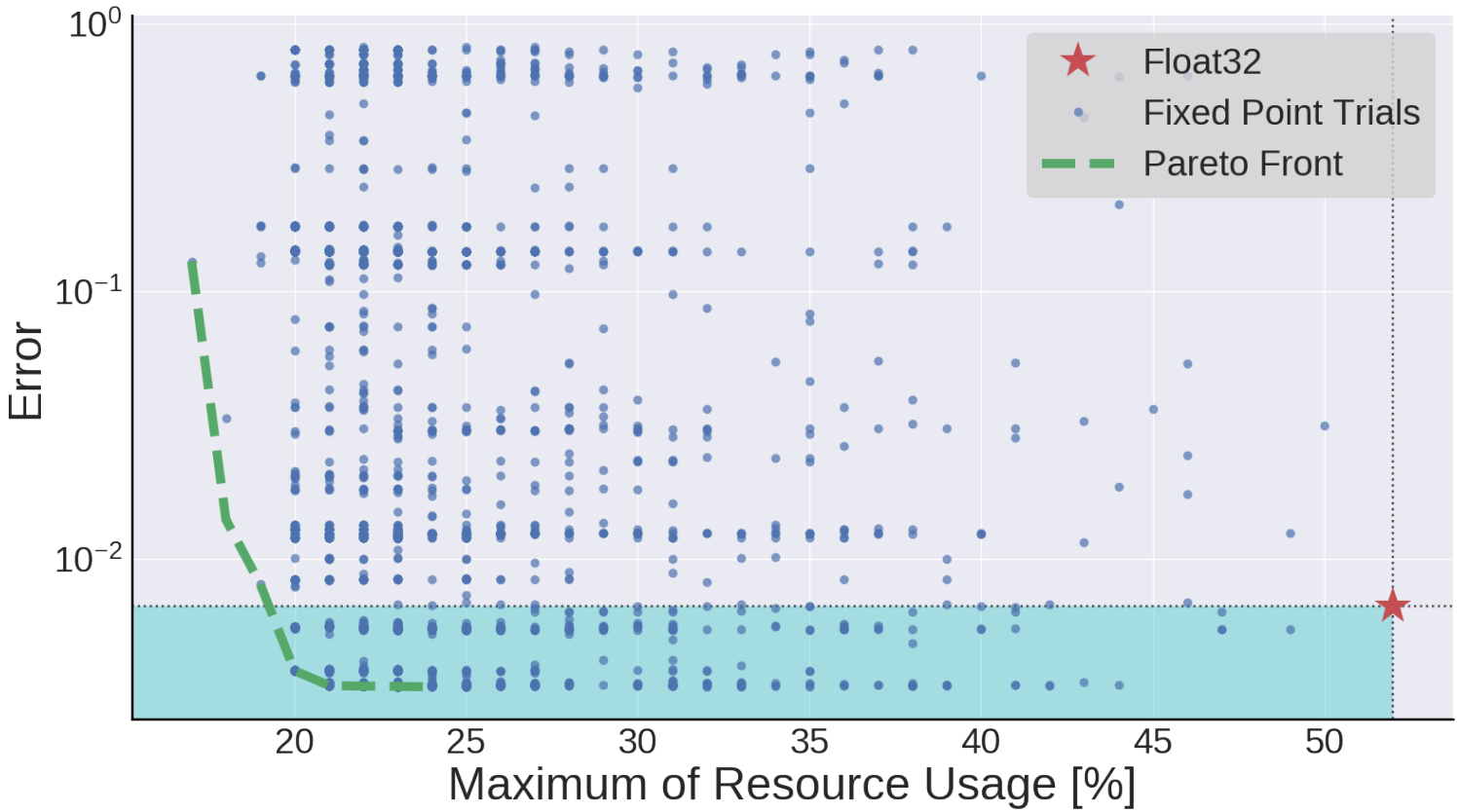
Let's look at some results:



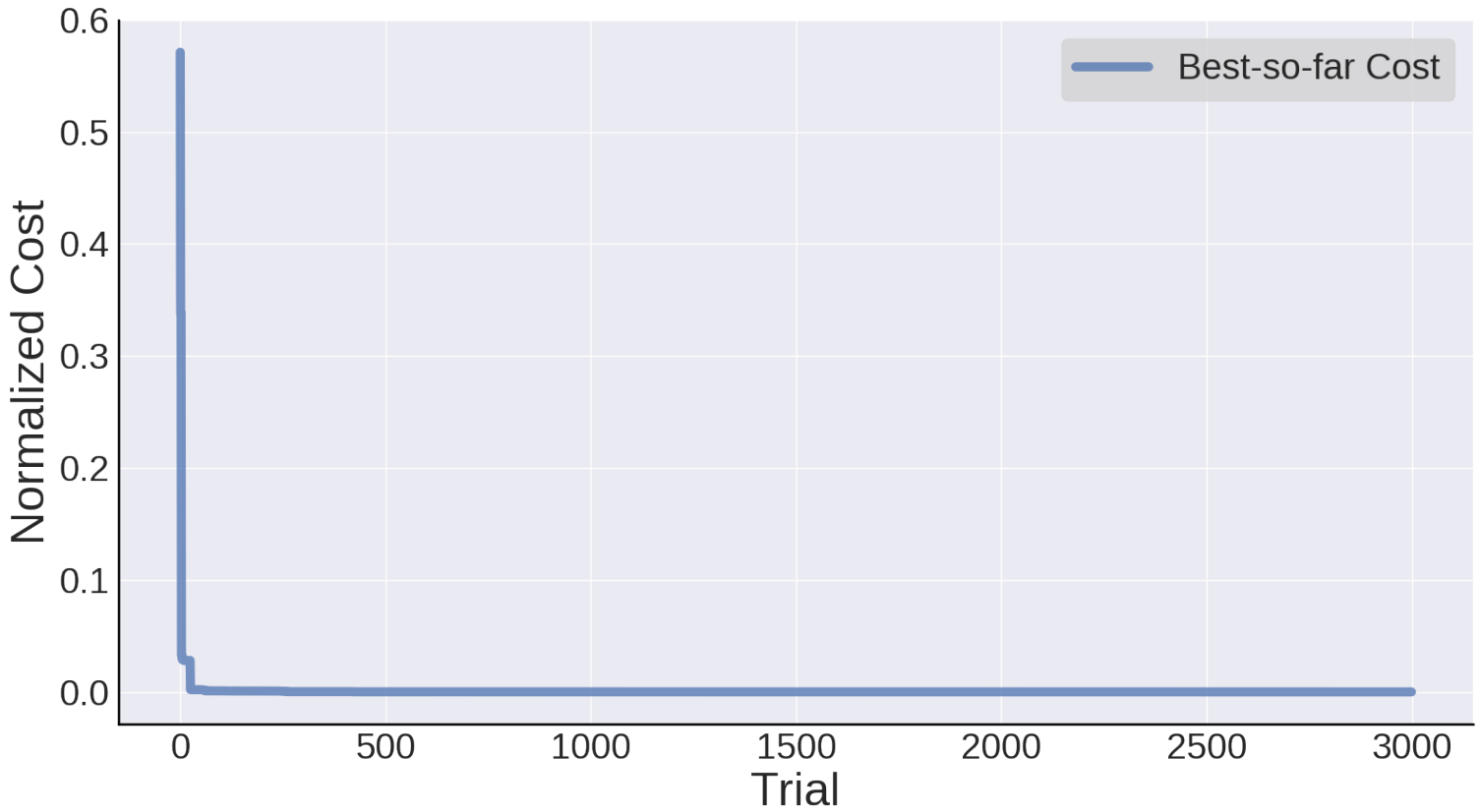
Let's look at some results:



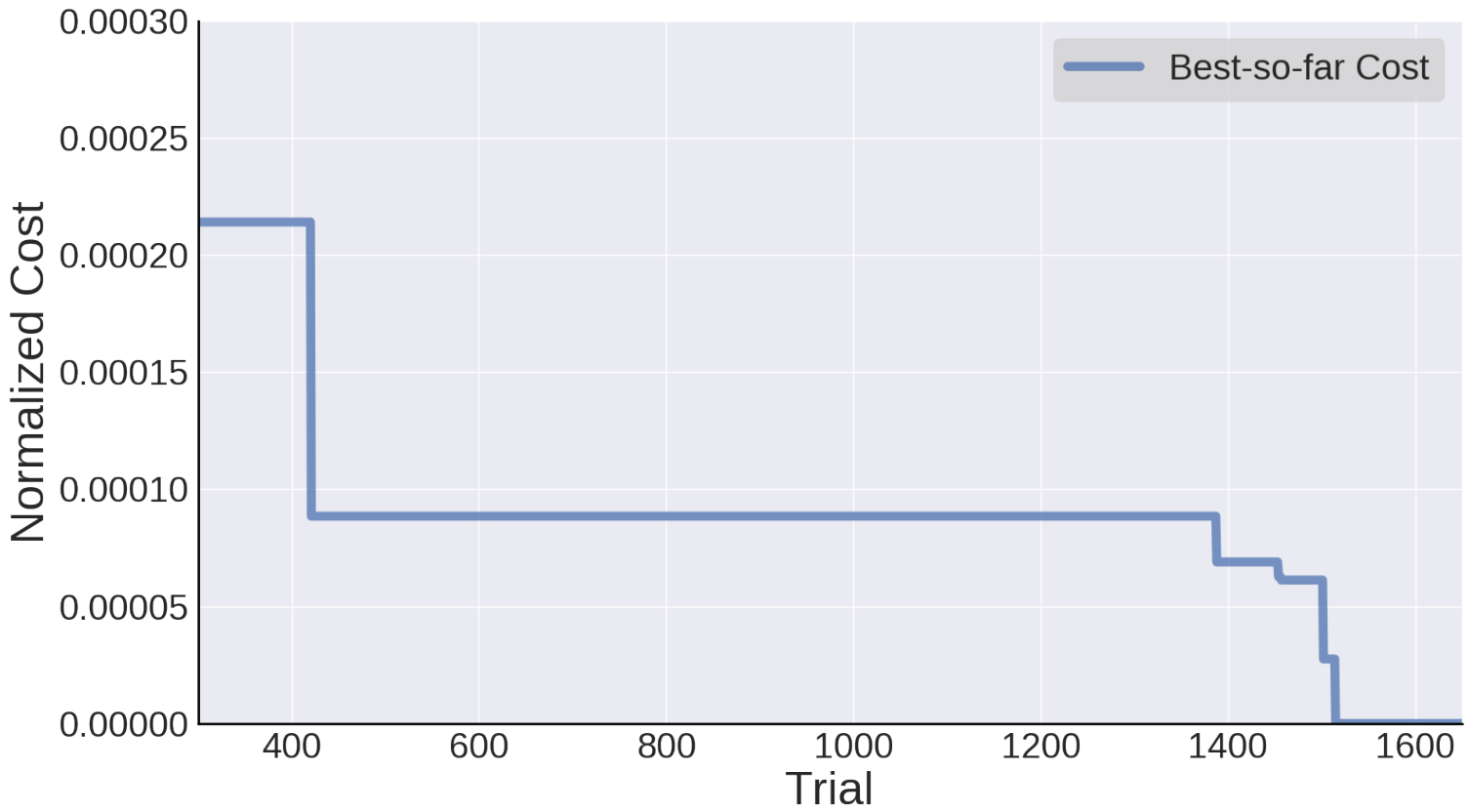
Let's look at some results:



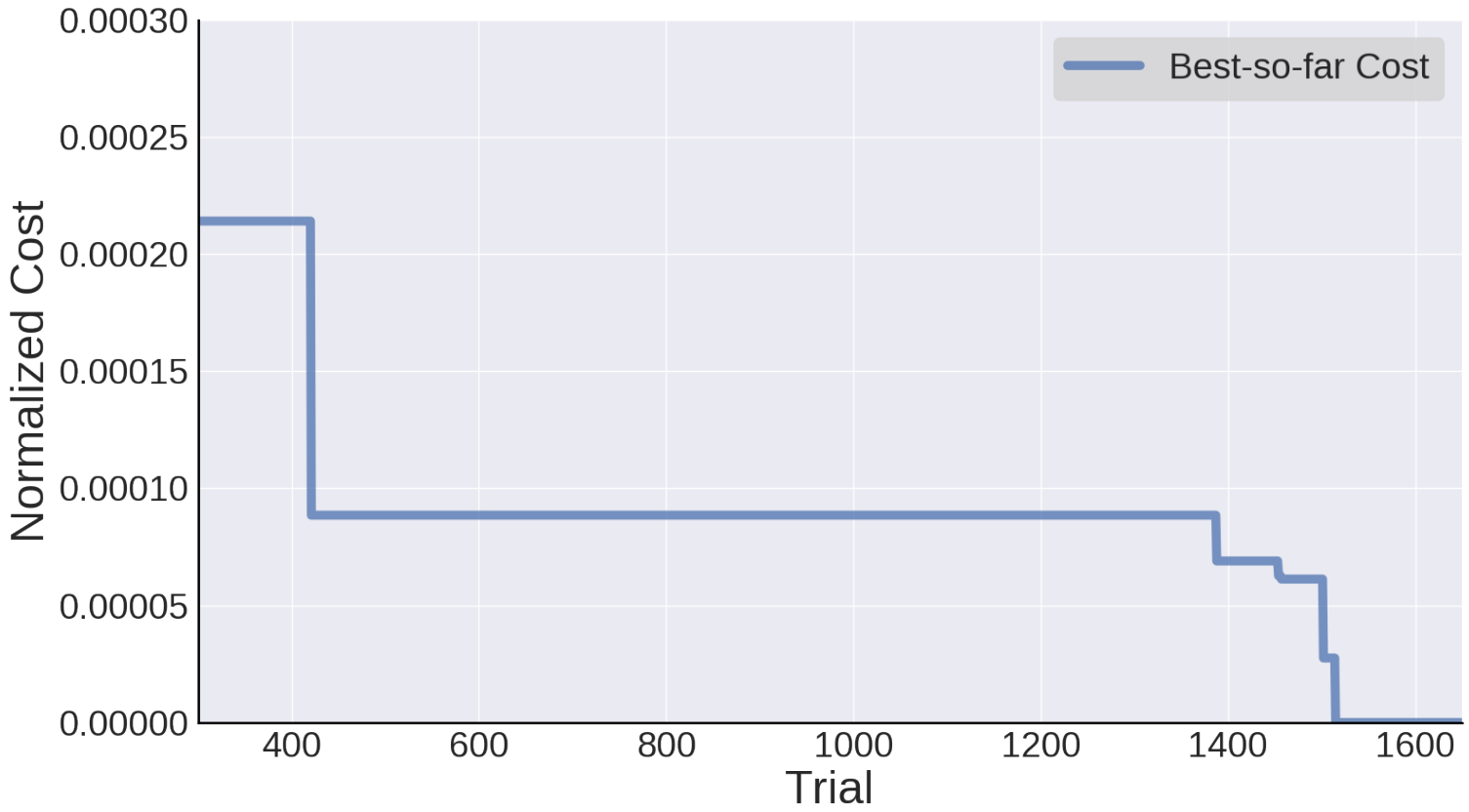
We let Hyperopt run for 3000 trials using a cost function that minimizes error and resources



In fact, Hyperopt converges to the best solution after 1500 trials



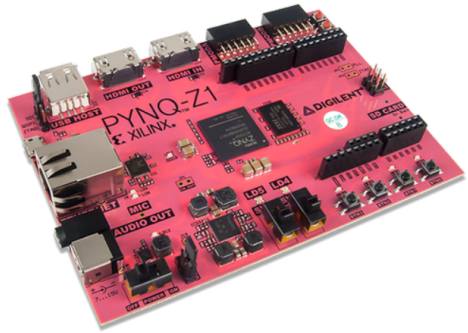
And, we are within 99% after ≈400 trials which takes 4 hours (i7-6700k)



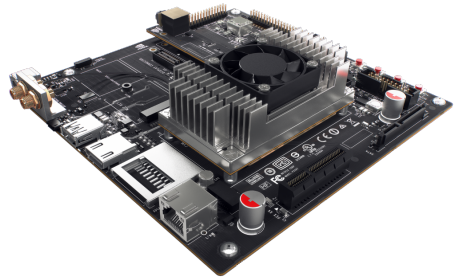
Results

Performance vs. Jetson TX1

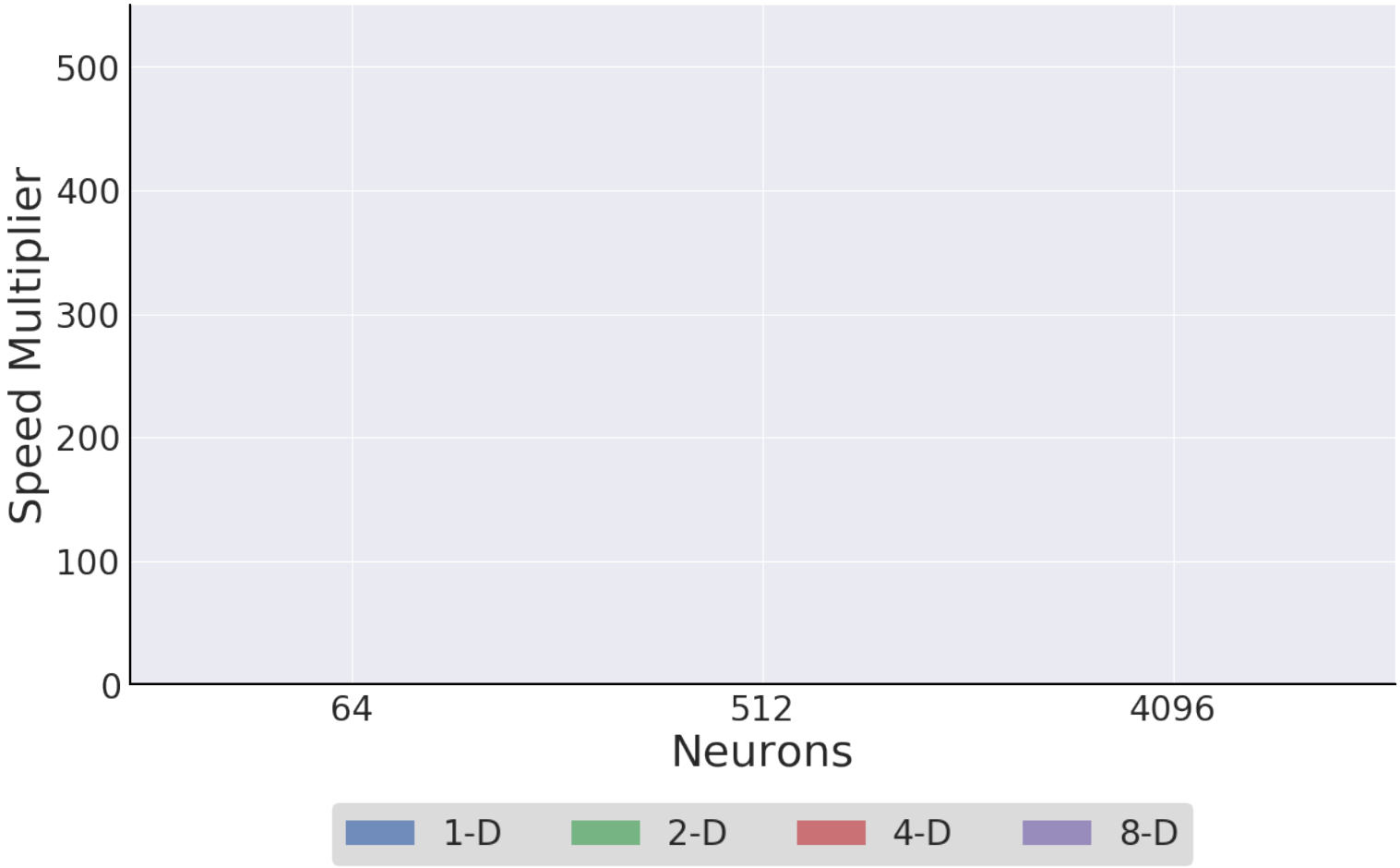
PYNQ uses a Zynq FPGA with 28nm technology.



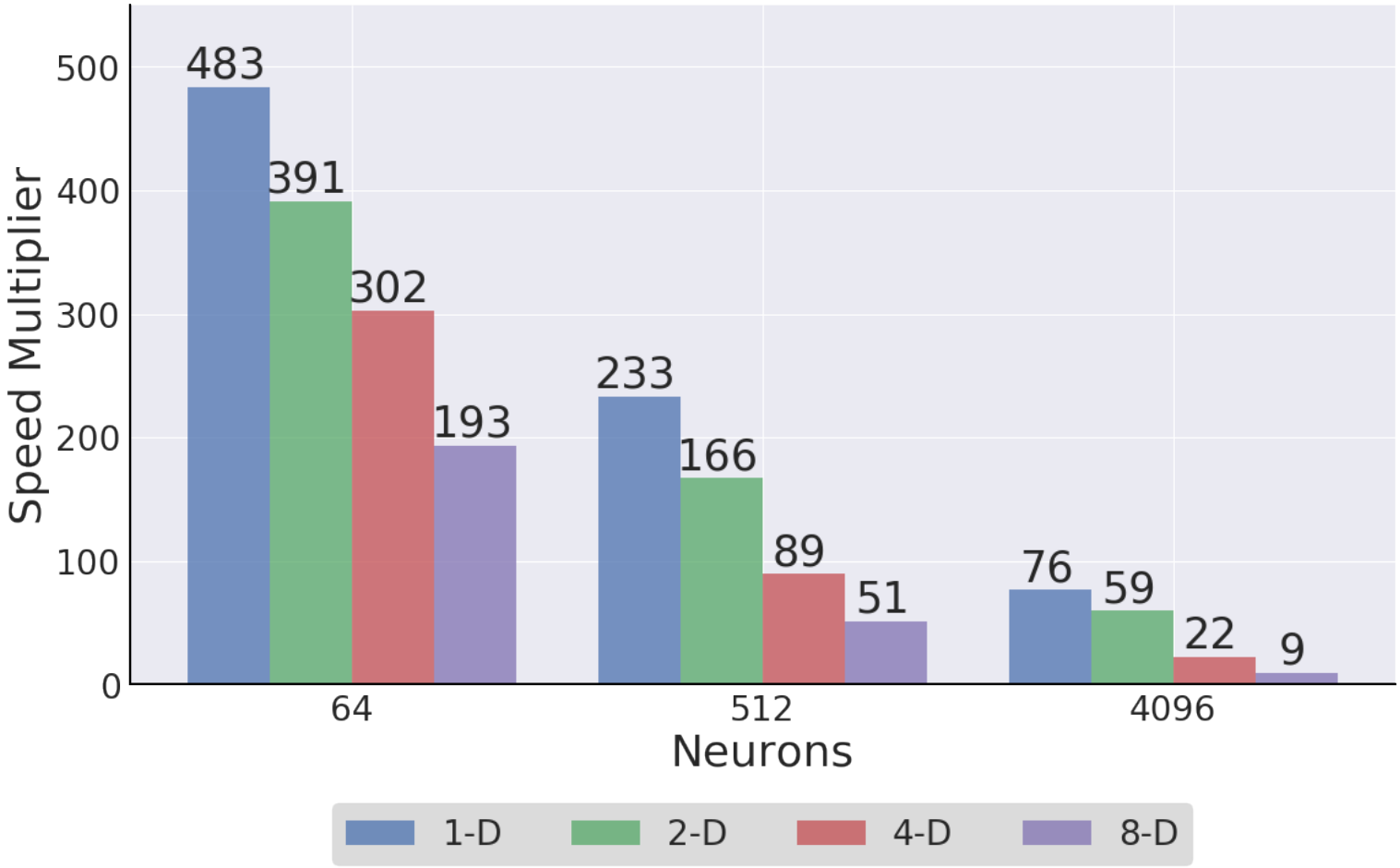
Jetson TX1 uses an embedded GPU with 20nm technology.



FPGA Speedup compared to cuBLAS implementation on Jetson TX1.



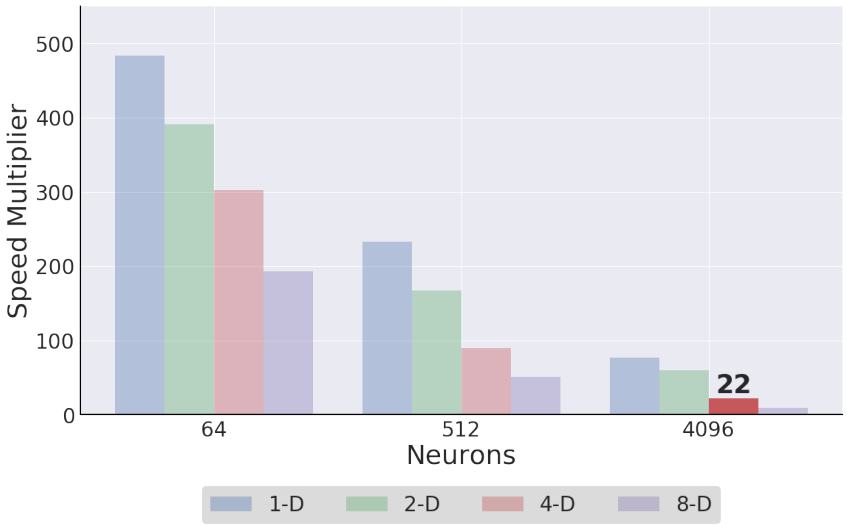
FPGA Speedup compared to cuBLAS implementation on Jetson TX1.



Practical Applications

Our NengoFPGA PYNQ implementation can accomplish useful work

3000 neurons (D=6) can form the basis for an adaptive motor controller [*DeWolf et al.*, 2016]



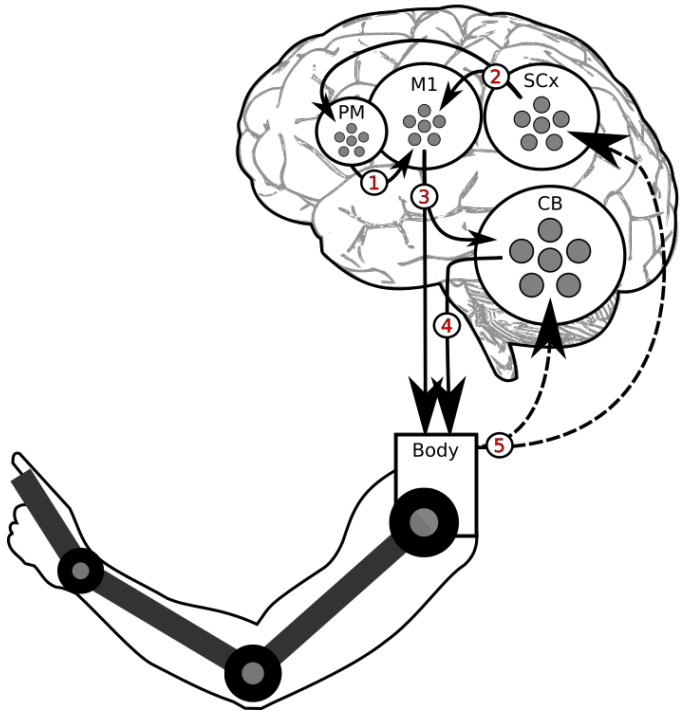
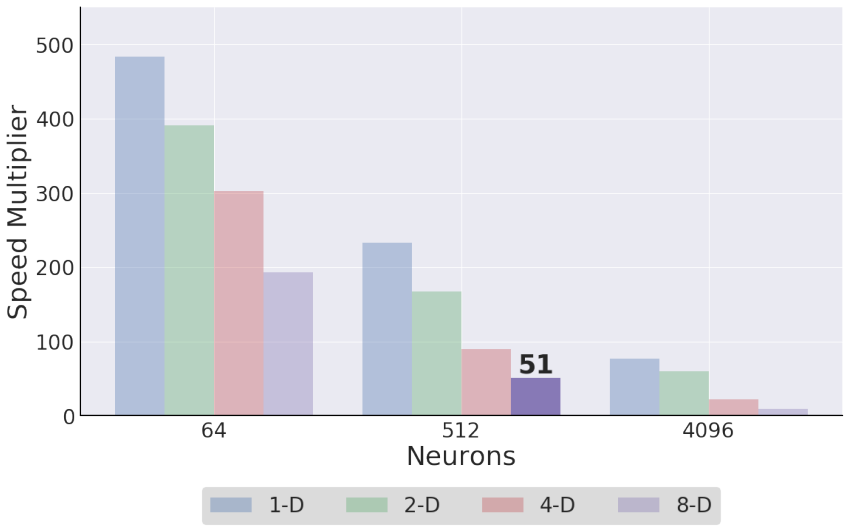


Image from “A spiking neural model of adaptive arm control” - DeWolf et al.

Practical Applications

500 neurons (D=30) can adapt and control a 15-joint body simulation [*Stewart et al.*, 2015]



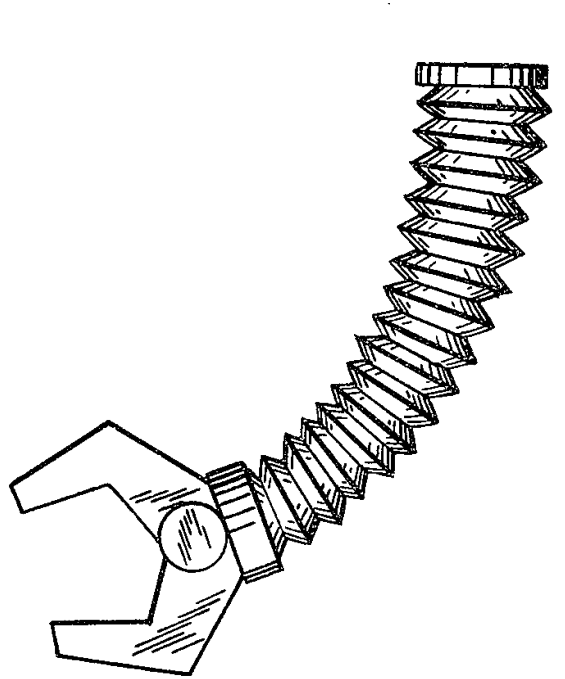
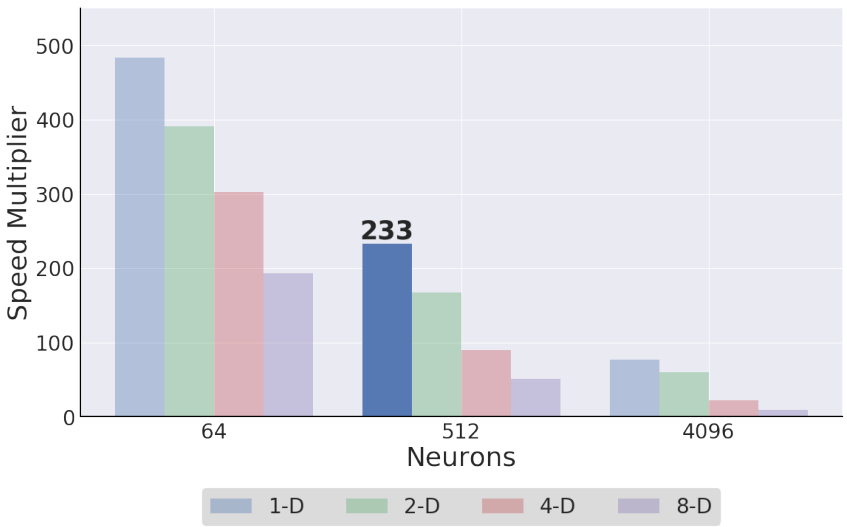


Image from Google

Practical Applications

We control a variable mass inverted pendulum with comparable accuracy to the floating-point design (N=1000, D=1)



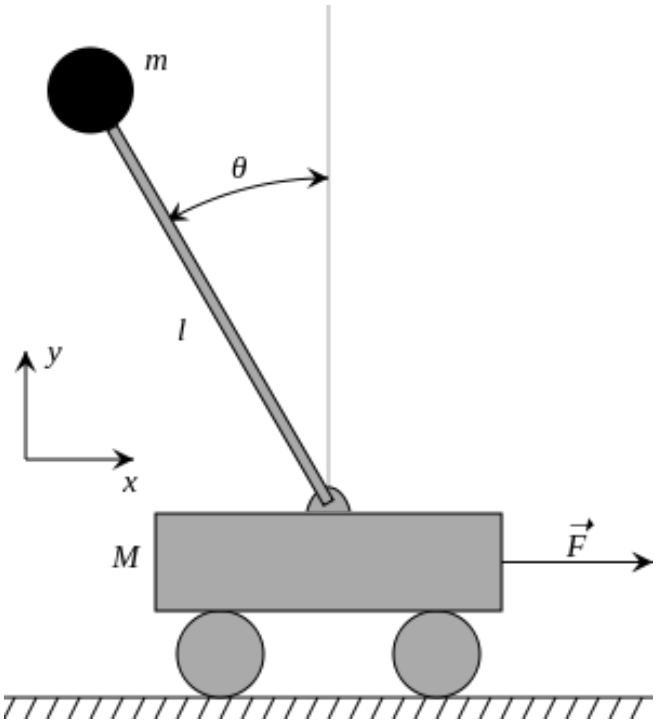


Image from Wikipedia

Conclusion

NengoFPGA is a user-friendly, efficient Python-accessible implementation designed for dynamic NEF style neural networks and cognitive modelling that outperforms comparable devices.

Conclusion

NengoFPGA is a user-friendly, efficient Python-accessible implementation designed for dynamic NEF style neural networks and cognitive modelling that outperforms comparable devices.

- Clever HLS architecture $\rightarrow$ 15$\times$ improvement

Conclusion

NengoFPGA is a user-friendly, efficient Python-accessible implementation designed for dynamic NEF style neural networks and cognitive modelling that outperforms comparable devices.

- Clever HLS architecture $\rightarrow$ 15$\times$ improvement
- Automatically tuned precision $\rightarrow$ 4$\times$ improvement

Conclusion

NengoFPGA is a user-friendly, efficient Python-accessible implementation designed for dynamic NEF style neural networks and cognitive modelling that outperforms comparable devices.

- Clever HLS architecture $\rightarrow$ 15$\times$ improvement
- Automatically tuned precision $\rightarrow$ 4$\times$ improvement
- 10–500$\times$ faster than Jetson TX1

Conclusion

NengoFPGA is a user-friendly, efficient Python-accessible implementation designed for dynamic NEF style neural networks and cognitive modelling that outperforms comparable devices.

- Clever HLS architecture $\rightarrow$ 15$\times$ improvement
- Automatically tuned precision $\rightarrow$ 4$\times$ improvement
- 10–500$\times$ faster than Jetson TX1
- 2–10$\times$ less power than Jetson TX1

Next Steps

We continue work to make NengoFPGA a seamless and fully featured backend for Nengo including:

- Multiple neuron types
- Larger networks
- Flexible connectivity
- Better CNN support

NengoFPGA is available for purchase from **Applied Brain Research** (url: `abr.rocks`)