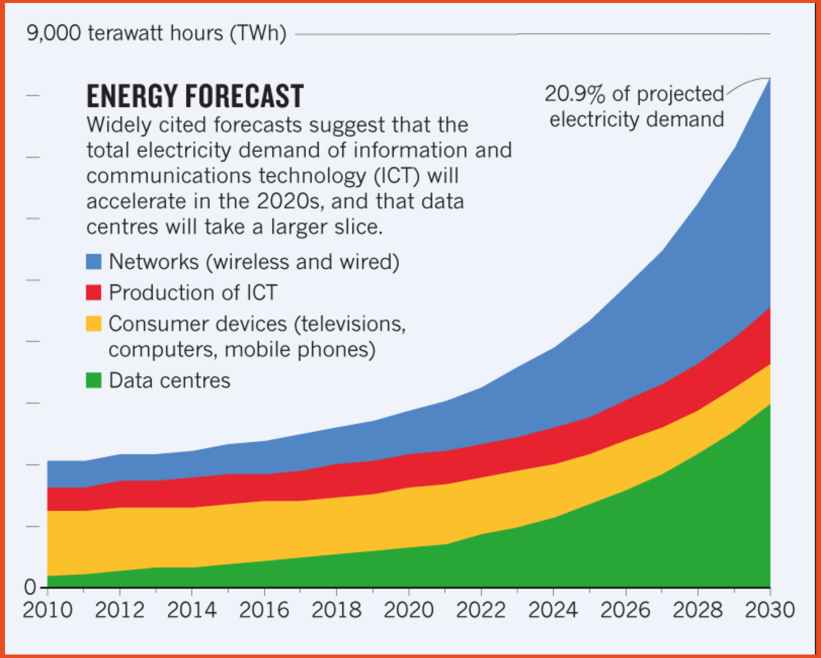
VISHAL BALAJI

VECTORIZATION & GPU-PARALLELIZATION

CONTENTS

INTRODUCTION



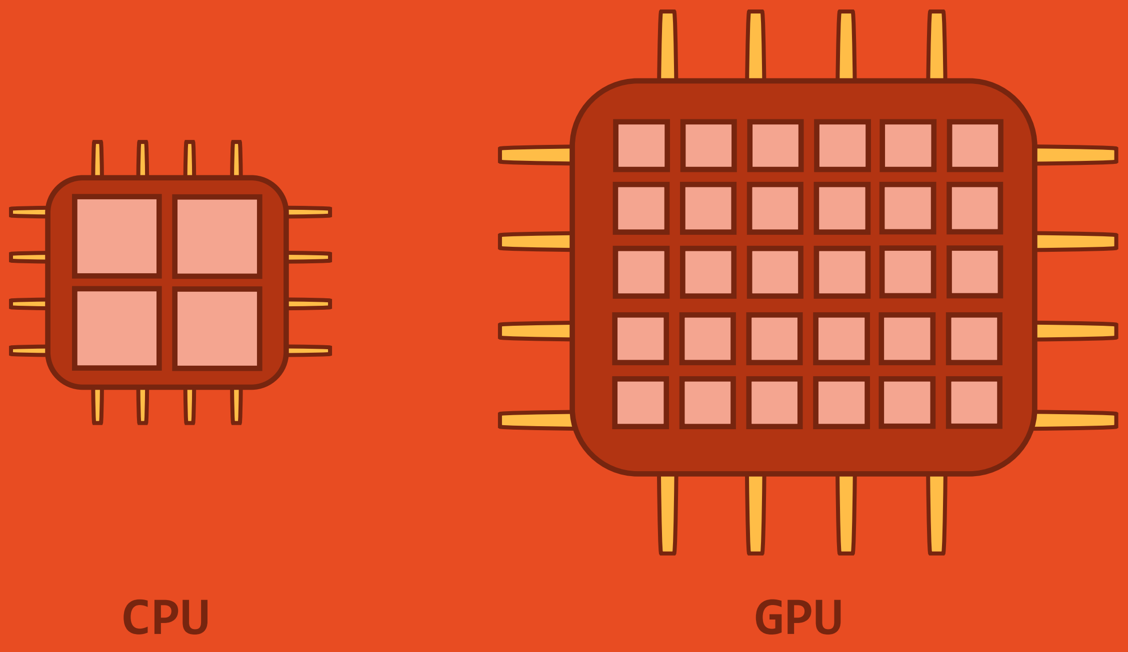
As of 2020, computing devices are responsible for 1% global CO_2 emissions. Expected to increase to 2.5% by 2030 [1]

Training OpenAI's ChatGPT used 1.287 Gwh, roughly equivalent to 120 US homes for a year [2]

Vectorization:

- Perform mathematical operations on entire arrays, instead of iterating over individual elements.
- Commonly used in computer programming and data processing to optimize performance and efficiency

INTRODUCTION



*Images not for scale

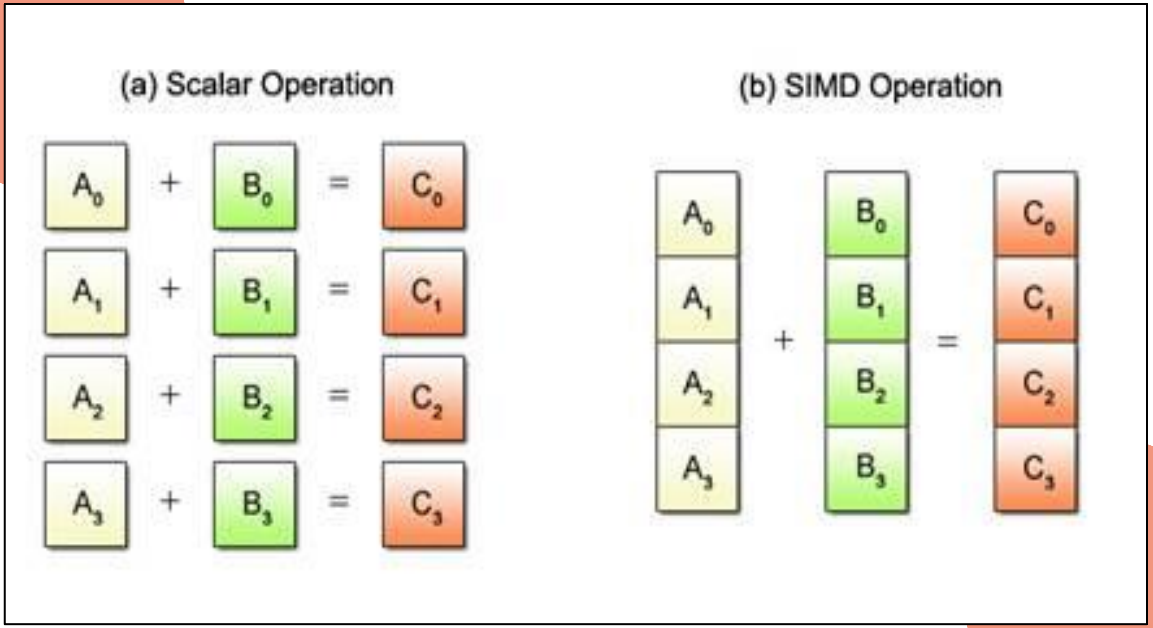
Central Processing Unit (CPU)

- Few (2-12) strong cores (each >4GHz)
- Suited for serial workloads
- Designed for general purpose calculations

Graphics Processing Unit (GPU)

- Thousands of weaker cores (each <2GHz)
- Suitable for parallel workloads
- Designed for graphics processing like 3D rendering

VECTORIZATION
SIMD COMPUTING



[3]

- SIMD – (Single Instruction/Multiple Data)
- Exploitating data-parallelism: Applying same operation to large amount of data in parallel
- Enables faster execution time/higher throughput by leveraging parallelism at the **instruction level**
- SIMD extensions like SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), NVPTX (Nvidia Parallel Thread Execution) are present in CPUs/GPUs

VECTORIZATION

SEQUENTIAL VS PARALLEL

Problem: Multiply element-wise the two arrays and add 6 to them
[21, 27, 10, 19, 18], [3, 8, 12, 19, 23]

Simple for-loop solution:

[21 17 10 19 18]
*
[3 8 12 13 23]
+6

69	222	126	253	420

Matrix multiply solution:

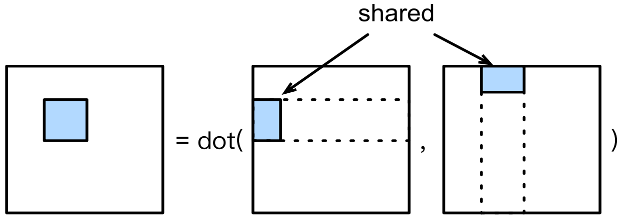
[21 17 10 19 18]
* * * * *
[3 8 12 13 23]
\+ \+ \+ \+ \+
6 6 6 6 6

69	222	126	253	420

VECTORIZATION

MATRIX ALGEBRA

WHY?



[4]

$$\begin{bmatrix} 1 & 2 \\ 7 & 5 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 9 \\ 7 & 5 & 3 \end{bmatrix} = \begin{bmatrix} 1x1 + 2x7 & 1x2 + 2x5 & 1x9 + 2x3 \\ 7x1 + 5x7 & 7x2 + 5x5 & 7x9 + 5x3 \end{bmatrix} = \begin{bmatrix} 15 & 12 & 15 \\ 42 & 39 & 78 \end{bmatrix}$$

DATA LOCALITY

- Accessing data close in memory
- Optimized cache utilization by accessing elements of matrices in specific pattern

DATA INDEPENDENCY

- Certain blocks of data being computed are independent of other elements
- Easily schedulable and mappable to SIMD units

INHERENTLY MATHEMATICS

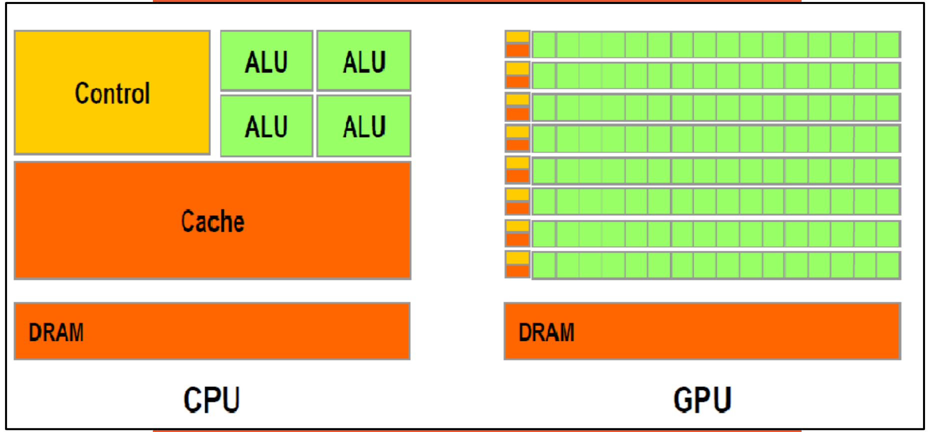
- Being used since 18th century
- Numerous algorithmic optimizations developed specifically for matrix algebra: Strassen algorithm, Coppersmith-Winograd algorithm

OPTIMIZED LIBRARIES

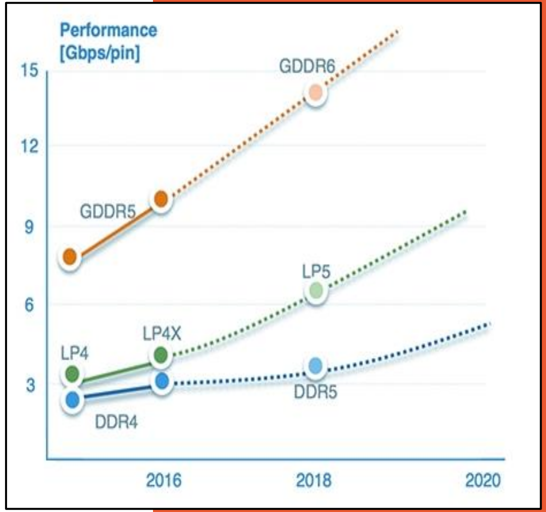
- Leveraging specific hardware specific features to provide efficient implementations
- Examples: openBLAS, Intel Math Kernel Library (MKL), cuBLAS

VECTORIZATION
GPU

- Implements SIMD architecture by default
- Thousands of cores for parallel execution | RTX 4090 has 16384 cores
- Memory Efficiencies:
  - More complex and interconnected cache system than CPU
  - Dedicated memory: Significantly faster than RAM
  - Aligned memory storage for faster access
- Extremely well optimized parallel computing languages: CUDA / OpenCL



[5]

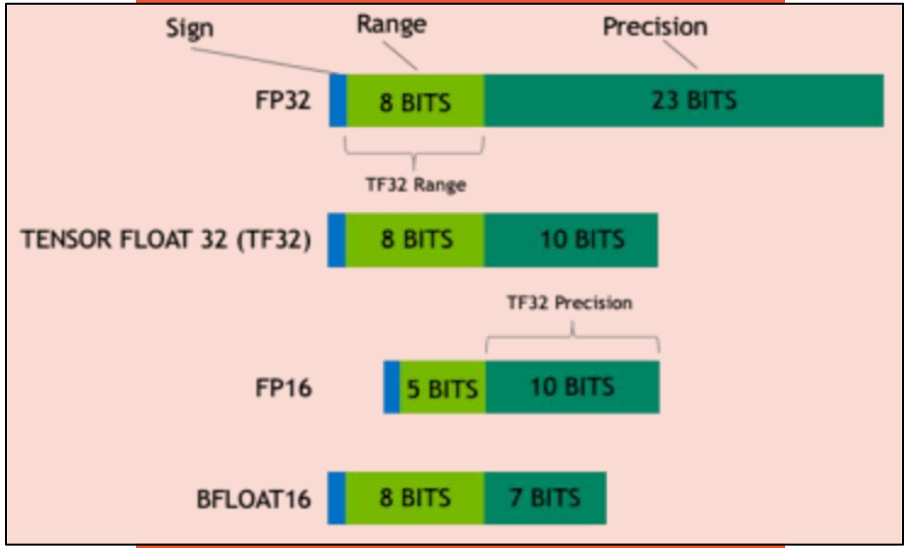


[6]

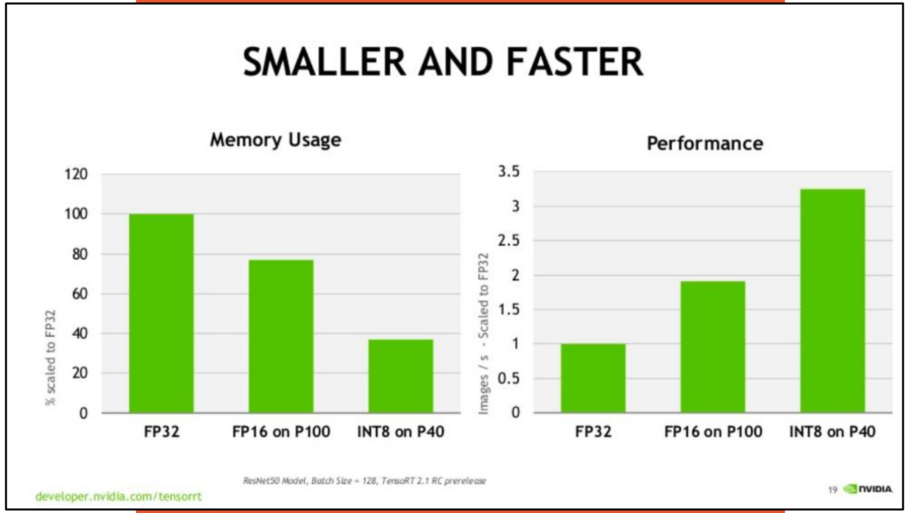
[7, 8]

VECTORIZATION
GPU

- Lower precision for futher speed up
  - GPUs compute default in **FP32** (originally for 3D applications)
  - For many math purposes (especially AI) **FP16** offers good enough precision
  - Close to 2x improvement in throughput
  - INT8, FP8, INT4 precision is also becoming increasingly used
- Specialized Hardware – Made exclusively for matrix calculations
  - Nvidia Tensor Cores
  - Intel Xe-Cores

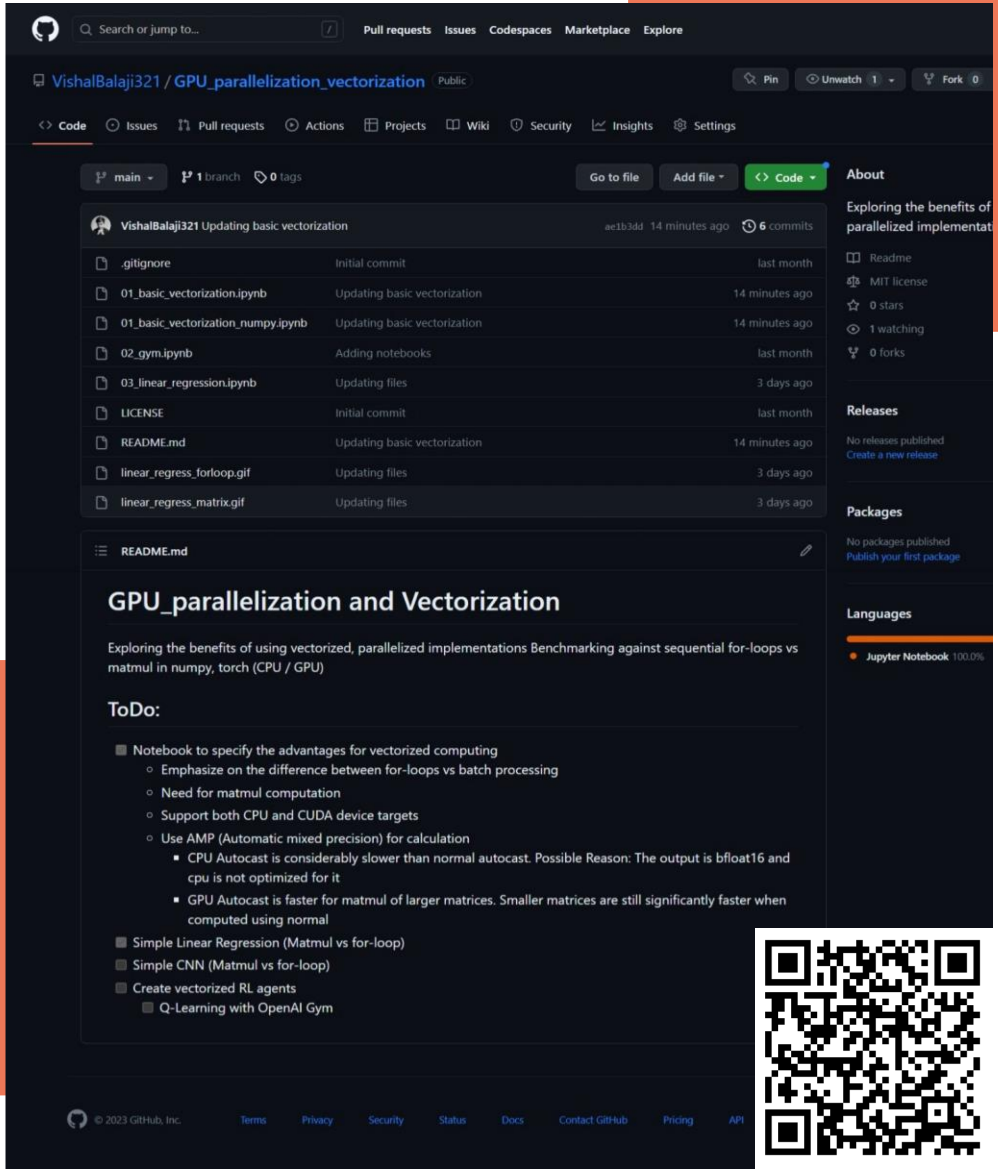


[9]



[10]

PROJECT
INTRODUCTION



- To show the performance advantage of using vectorization and performance gains in using GPUs
- Primarily comparing normal for-loops (sequential) with matrix multiplication in following cases:
  - Dot Product **(1-D and N-Dimensional array)**
  - Machine Learning: **Linear Regression**
  - Deep Learning: **CNN**
  - Reinforcement Learning: **Q-Learning**

PROJECT

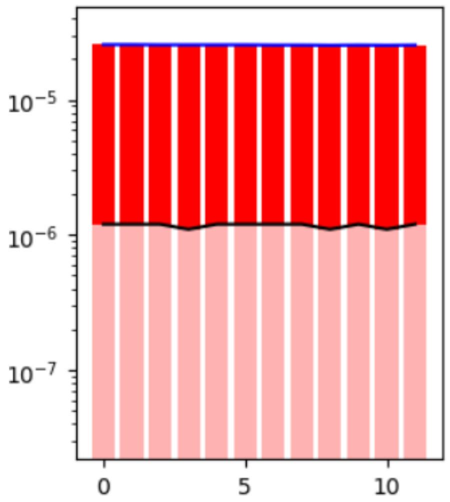
DOT PRODUCT

$$\begin{bmatrix} 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 7 \end{bmatrix} = 2*1 + 3*7 = 23$$

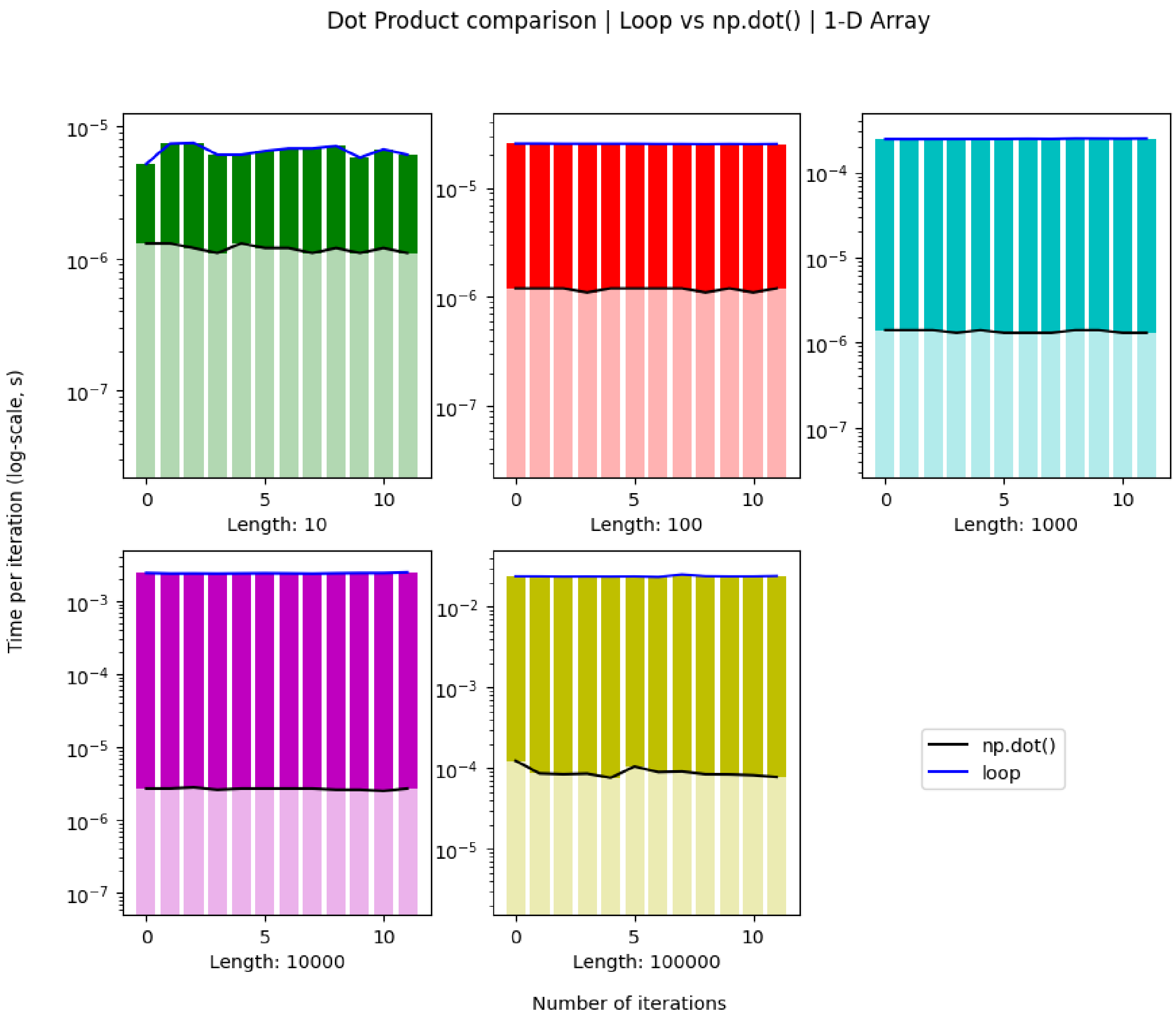
```python
for _ in range(NUMBER_ITERS): # Doing same computation for 20 times to reduce runtime variance
    dot_loop_value = 0.0
    t_loop_start = perf_counter()
    for index in range(a.shape[0]):
        dot_loop_value += a[index] * b[index]
    t_loop_end = perf_counter()

    t_loop_total = t_loop_end - t_loop_start
    if _ > IGNORE_ITERS: # Ignoring first 3 loop iterations, to further reduce the noise
        avg_loop_time_list.append(t_loop_total)
avg_loop_time_list = np.array(avg_loop_time_list)
avg_loop_time = np.mean(avg_loop_time_list)
```

Sample code for benchmarking (For-loop, 1D array)



- What is dot product?
- To observe performance benefit of matrix dot product in comparison to for-loops based dot product at different sizes
- Done using Python numpy package for CPUs and PyTorch for GPUs
- Calculating speed by doing same dot product 15 times to eliminate variance (also ignoring first 2 computations)

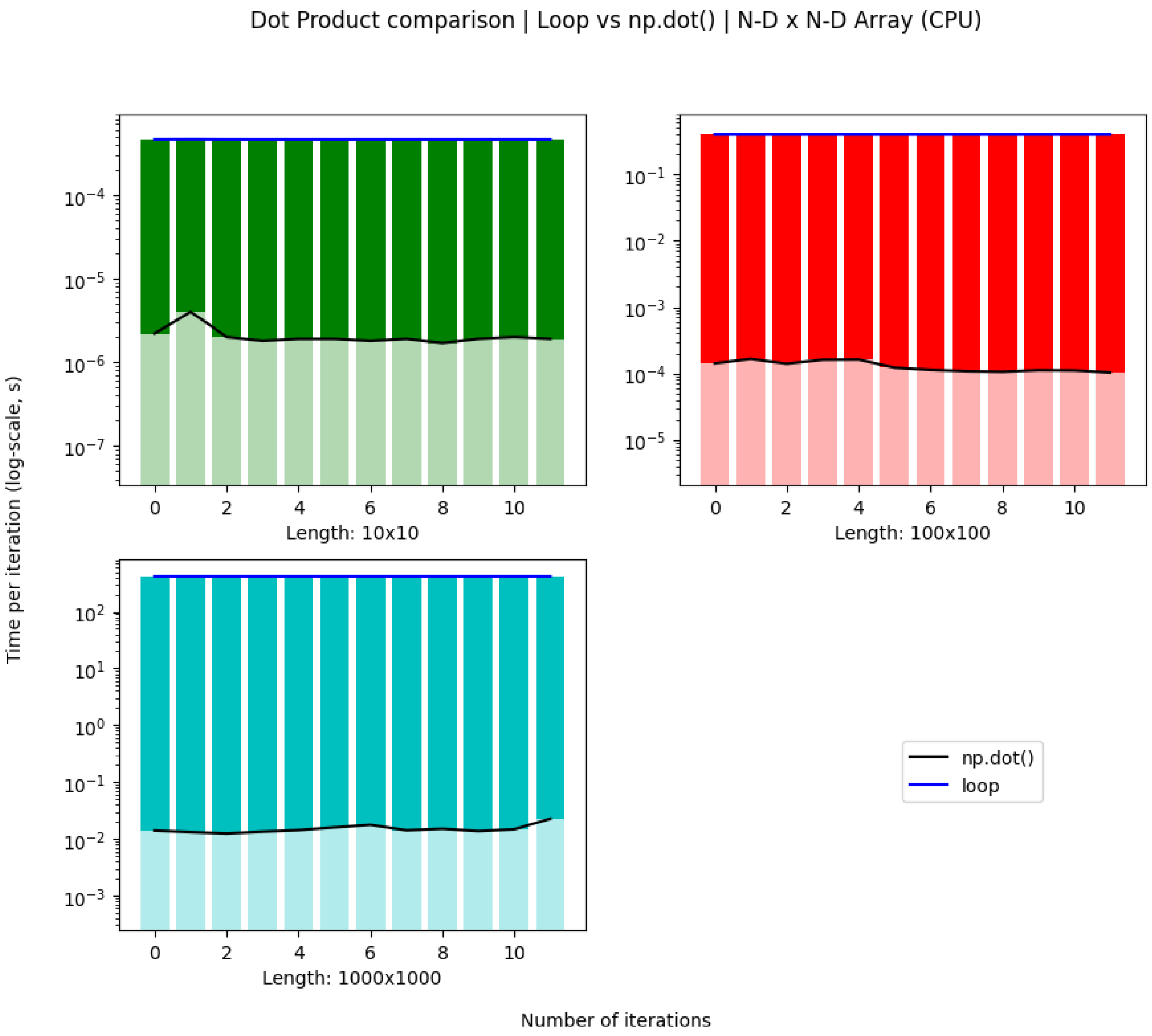


PROJECT

DOT PRODUCT – 1D ARRAY

Length of array	Time required (in µs)		SpeedUp (in x)
	Loop	np.dot()	
10	5.3	1.2	4.4
100	24	1.2	20.6
1000	246	1.3	182.1
10000	2420	2.7	906.1
100000	23700	88.5	267.8

- SpeedUp: time for Loop / time for np.dot()
- Operation: (1-D,) @ (1-D,)



PROJECT

DOT PRODUCT - ND ARRAY CPU

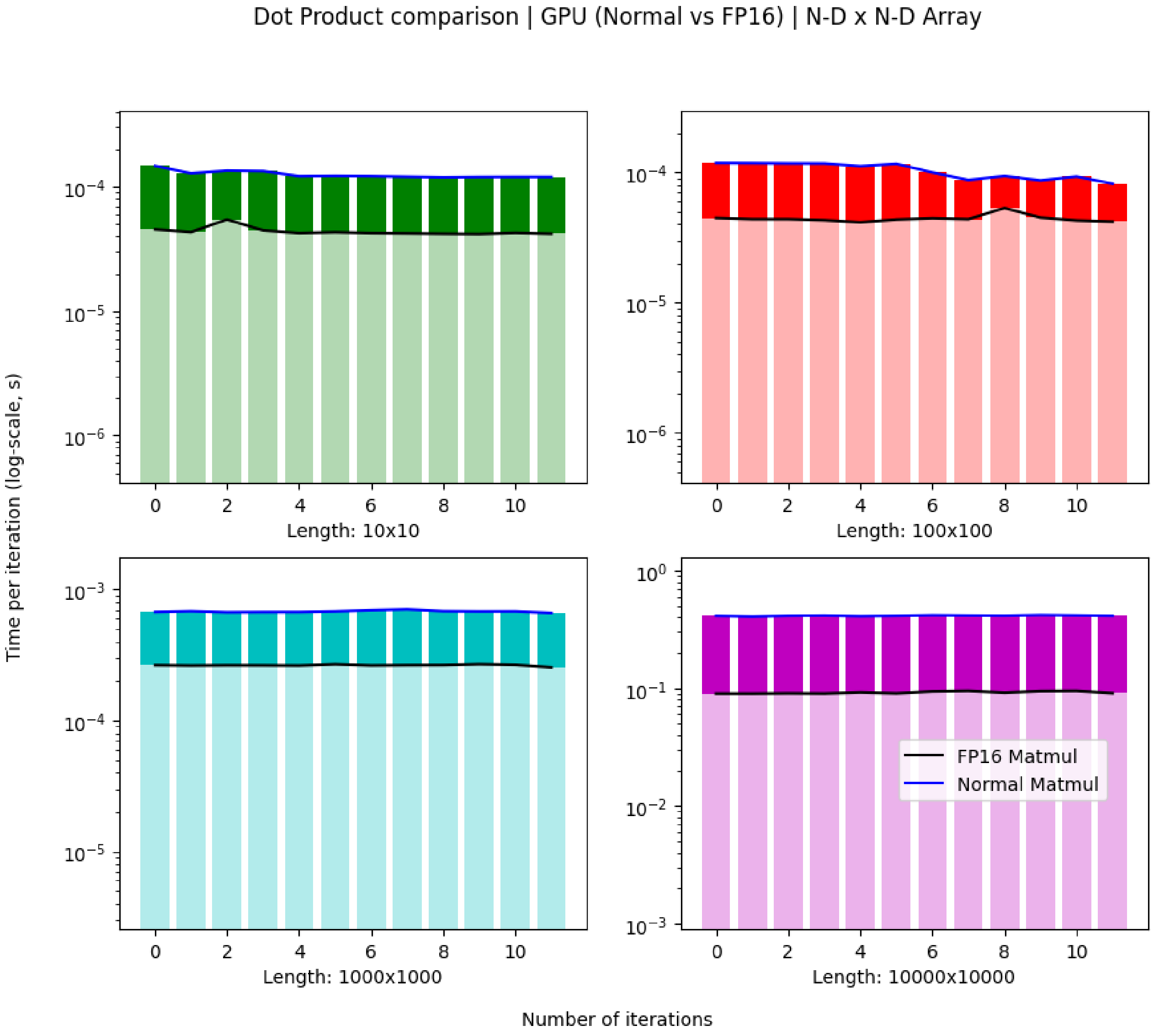
Shape of array	Time required (in ms)		SpeedUp (in x)
	Loop	np.dot()	
10,10	0.45	0.002	219
100,100	40.4	0.1	3153
1000,1000	417000	15	27731

- Operation: (N-D, N-D) @ (N-D, N-D)

```
result = []
t_loop_start = perf_counter()
for i in range(a.shape[0]):
    row = []
    for j in range(b.shape[1]):
        product = 0
        for k in range(a.shape[1]):
            product += a[i][k] * b[k][j]
        row.append(product)
    result.append(row)
```

PROJECT

DOT PRODUCT - ND ARRAY GPU



Shape of array	Time required (in ms)		SpeedUp (in x)
	Matmul (FP32)	Matmul (FP16)	
10,10	0.082	0.043	1.87
100,100	0.059	0.044	1.34
1000,1000	0.41	0.26	1.58
10000,10000	322	92	3.5

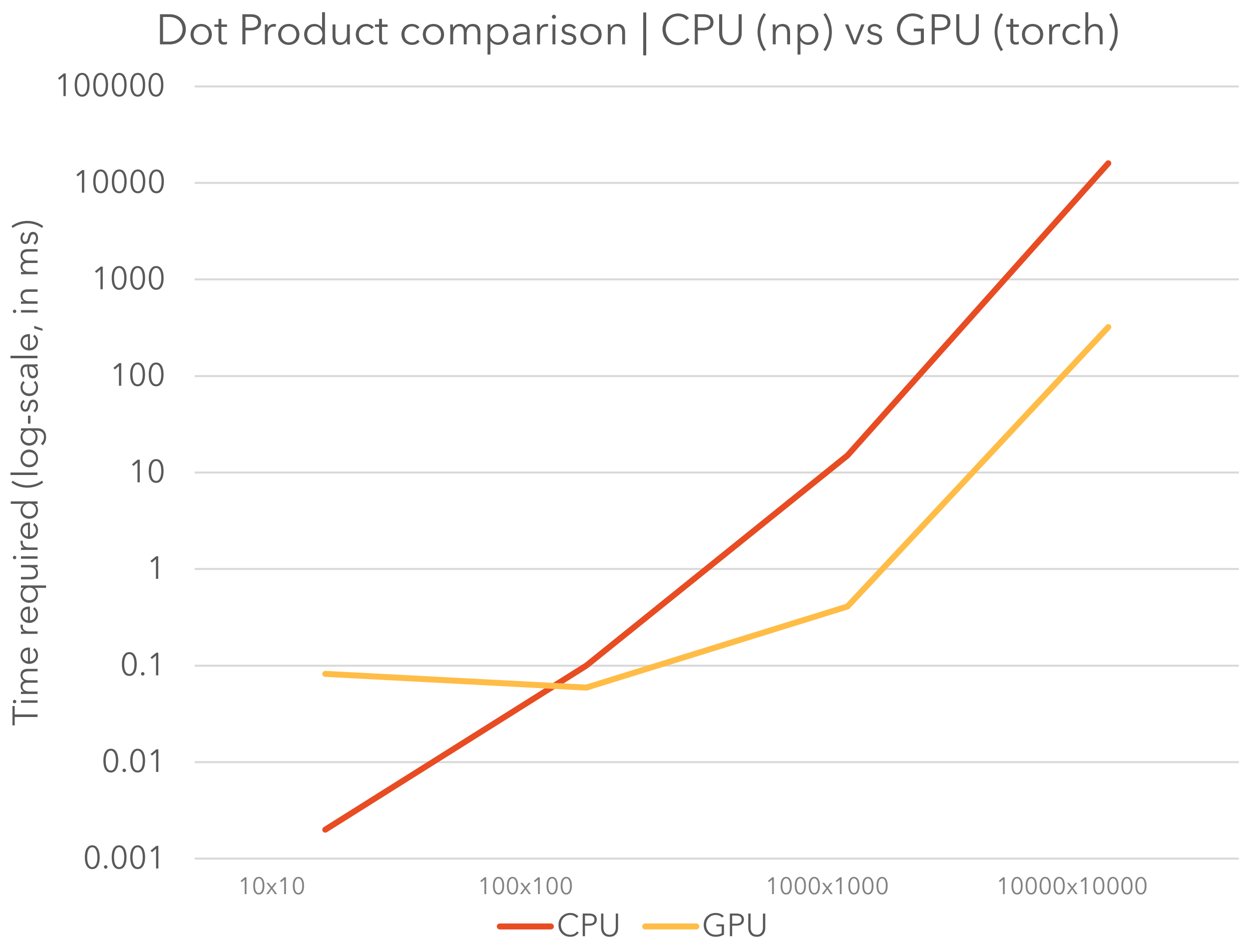
- Using PyTorch for GPU processing
- FP16/32 – Floating point 16/32 bits
- Operation: (N-D, N-D) @ (N-D, N-D)

PROJECT

DOT PRODUCT - ND ARRAY
GPU VS CPU

Shape of array	Time required (in ms)		SpeedUp (in x)
	Matmul (GPU)	Matmul (CPU)	
10,10	0.082	0.002	0.024
100,100	0.059	0.1	1.69
1000,1000	0.41	15	36.5
10000,10000	322	16000	3.5

- GPU is significantly faster than CPU for larger arrays
- Why not small arrays?
  - For every GPU matmul, data must be first copied to GPU and then computed. This data-copy has time penalty and dominates while computing small arrays,



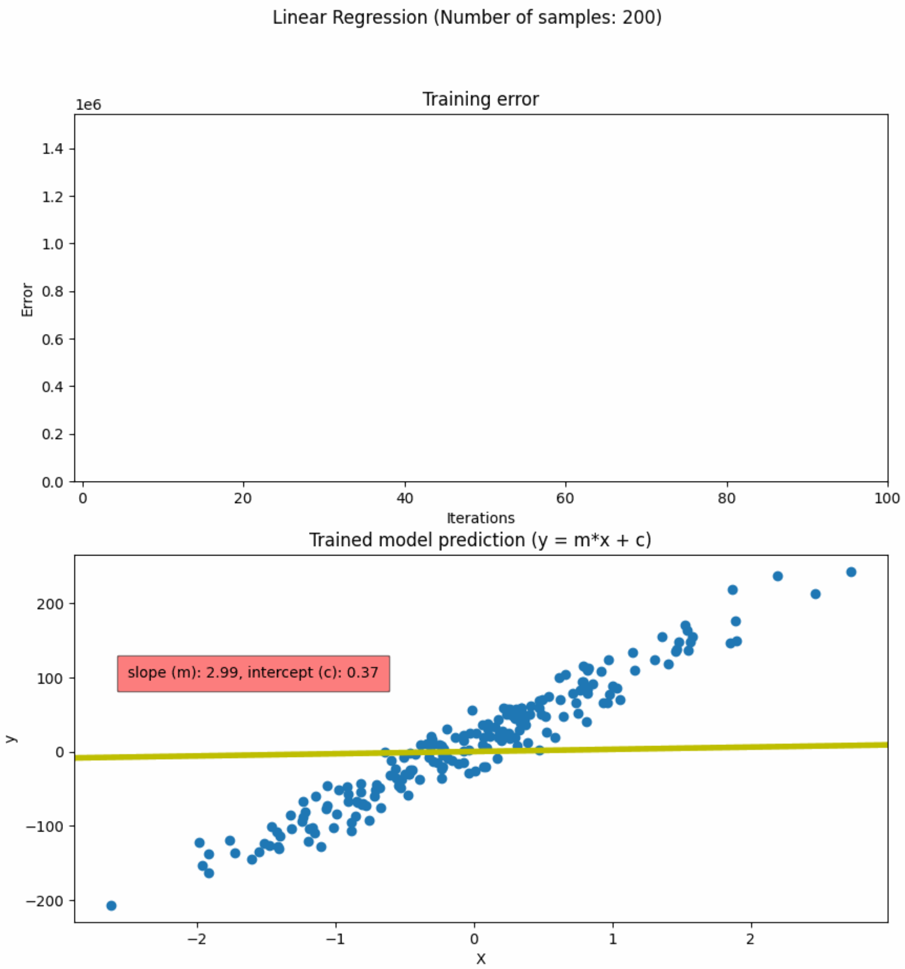
PROJECT

ML: LINEAR REGRESSION

- Finding the right variables that fit the equation: $y = mx + c$
- Using Mean Squared Error (MSE) as cost function

$$MSE = \frac{1}{n}\sum_{i=1}^{m}\left(y_i - (mx_i + c)\right)^2$$

- Using gradient descent to update the variables:
  - $m = m - \alpha \cdot \frac{2}{m}\Sigma(\hat{y} - y)$
  - $c = c - \alpha \cdot \frac{2}{m}\Sigma(\hat{y} - y).x$
- Using Learning rate: 0.0001 for 100 epochs



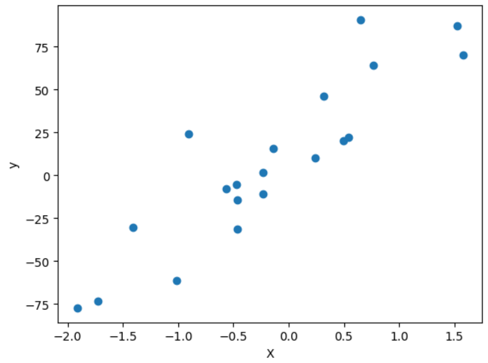
PROJECT

ML: LINEAR REGRESSION | RESULTS

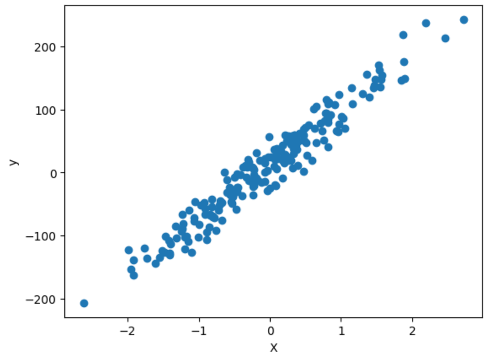
Number of sample	Time required / epoch (in s)		SpeedUp (in x)
	Loop	Matmul	
20	0.005	1.17E-5	427
200	0.04	1.32E-5	3030
500	0.25	2.52E-5	9920
1000	1	3.28E-5	30500
2000	4.2	3.65E-5	115000

**All computations executed in CPU*

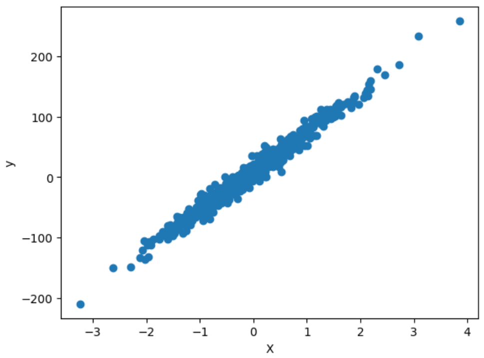
- Using sklearn.datasets to generate regression datasets with some noise



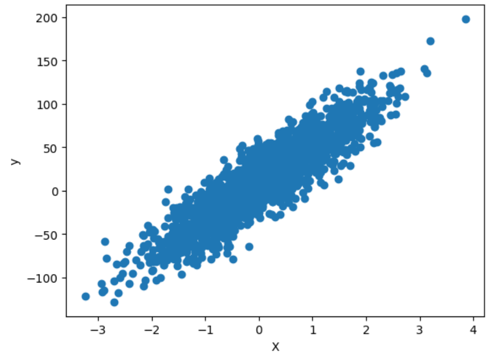
Number of samples: 20



Number of samples: 200



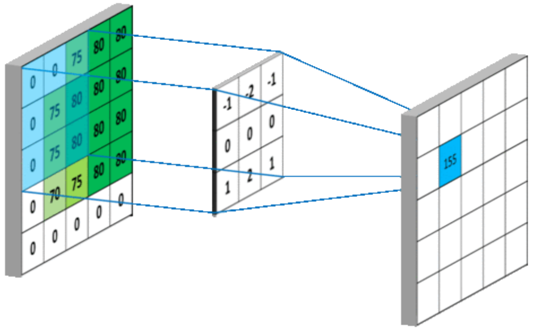
Number of samples: 500



Number of samples: 2000

PROJECT
DEEP LEARNING: CNN

- Simple Convolution Neural Network (CNN) Classifier
- Dataset: CIFAR-10
- Stochastic Gradient Descent (SGD) with Cross Entropy Loss
- Training with Mixed Precision (FP16)
- **Batchsize:** Number of training samples processed in a single forward and backward pass
    - Efficiency: Faster parallel processing
    - Memory Utilization: Processing the data in batches reduces the memory requirements compared to processing one example at a time
    - Stability: Updating NN with multiple data points reduces variance



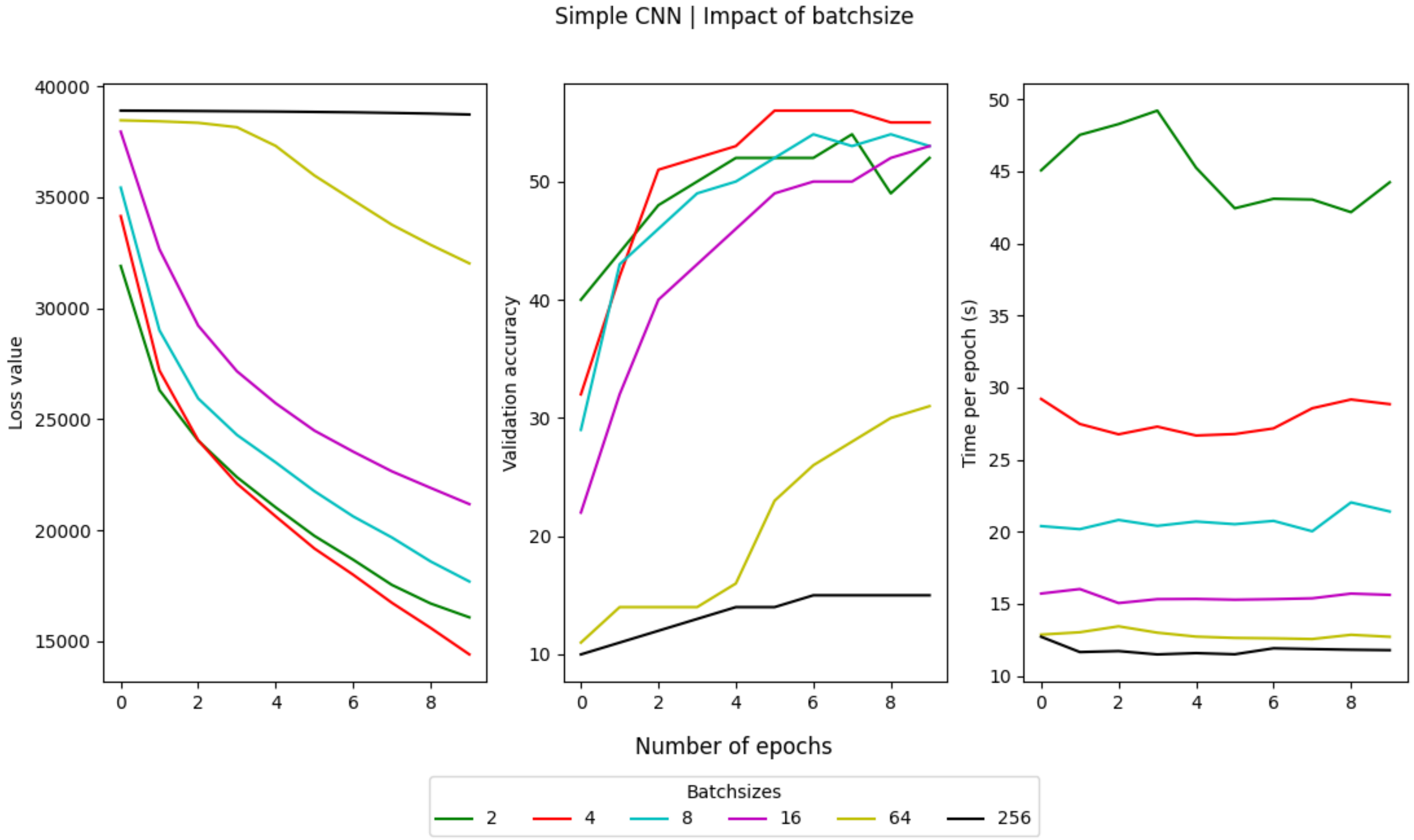
Working of CNN [14]

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

CIFAR 10 Dataset [11]

PROJECT

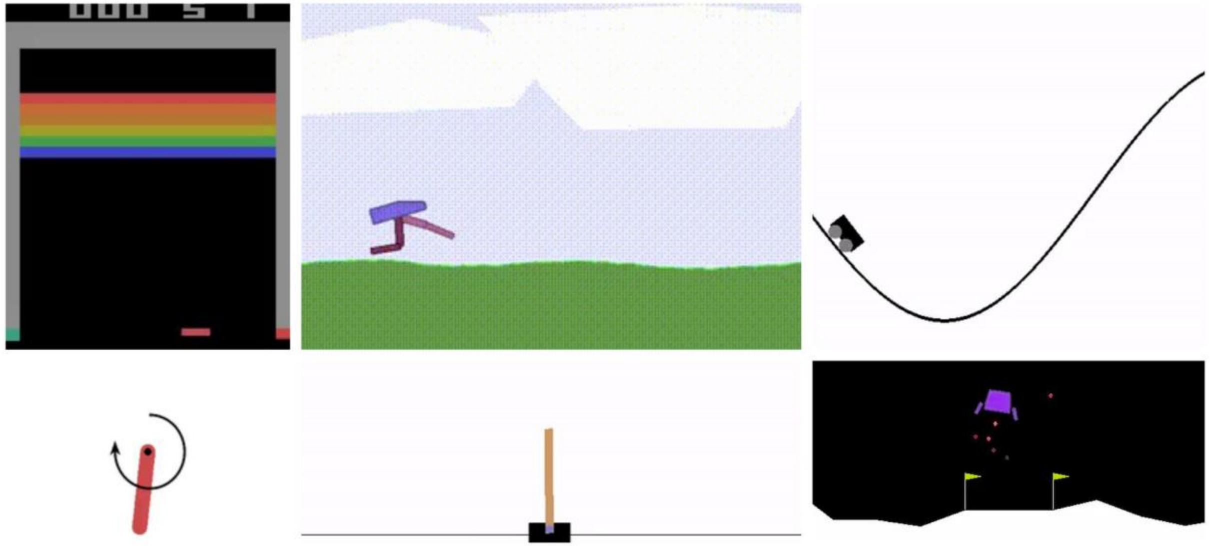
DEEP LEARNING: CNN | RESULTS



Batchsize	Min. Loss	Max. Validation Accuracy (%)	Time per epoch (s)
2	16085	54	**45.0**
4	**14416**	**56**	27.8
8	17700	54	20.7
16	21182	53	15.4
64	32020	31	12.8
256	**38722**	**15**	**11.8**

PROJECT

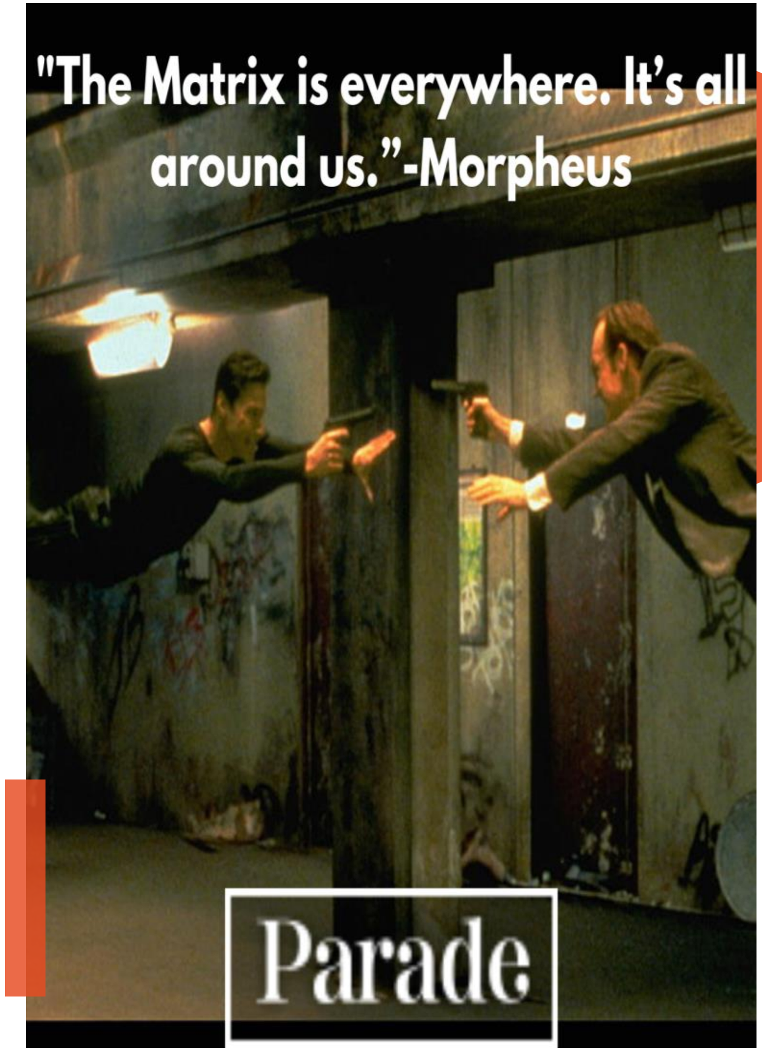
IMPROVEMENTS / TODO



OpenAI Gym Environments [12]

- **RL: Implement Vectorized Q-Learning Agent (with OpenAI Gym)**
- **Bug:** Check why Matmul is slower with Linear Regression in GPU
- **Improvement:** Display 1% low and 95% quantile for benchmark result
- **Improvement:** Polish the notebooks, more clear documentation/Readme file and add references

CHALLENGES WITH VECTORIZATION



This is a meme [13]

- Can be complex to formulate
- Not all for-loop can be converted to matrix math:
  - Example: Conditional statements like if, else and Search operations like np.where()
  - Loop-unrolling methods are effective for above situations: *np.vectorize()*, *tf.vectorized_map()*, *torch.vectorize()*
- If any iteration of the loop depends on value from previous iteration, vectorization is close to impossible
- Scheduling overhead must be considered

FOR LOOPS ARE NOT BAD !!

THANK YOU !!

Copy of my presentation

REFERENCES

[1] The Amount of Data Center Energy Use - AKCP Monitoring

[2] ChatGPT's energy usage is a mystery - here's what we do know (techhq.com)

[3] Basics of SIMD Programming (cvut.cz)

[4] 4. Matrix Multiplication — Dive into Deep Learning Compiler 0.1 documentation (d2l.ai)

[5] CPU vs GPU | Definition and FAQs | HEAVY.AI

[6] What's the Difference Between GDDR and DDR Memory? | Exxact Blog (exxactcorp.com)

[7] OpenCL Overview - The Khronos Group Inc

[8] Overview — CuPy 12.0.0 documentation

[9] FP64, FP32, FP16, BFLOAT16, TF32, and other members of the ZOO | by Grigory Sapunov | Medium

[10] Benchmarking GPUs for Mixed Precision Training with Deep Learning (paperspace.com)

[11] CIFAR-10 and CIFAR-100 datasets (toronto.edu)

[12] Getting Started With OpenAI Gym | Paperspace Blog

[13] Matrix Quotes: Neo, Morpheus, Agent Smith And Trinity Quotes - Parade: Entertainment, Recipes, Health, Life, Holidays

[14] Overview of Convolutional Neural Network in Image Classification (analyticsindiamag.com)

QUESTIONS/FEEDBACK