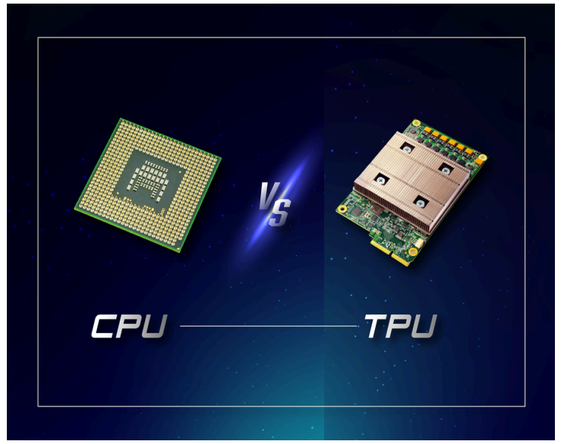
26-05-2025

REPRODUCIBLE RESEARCH

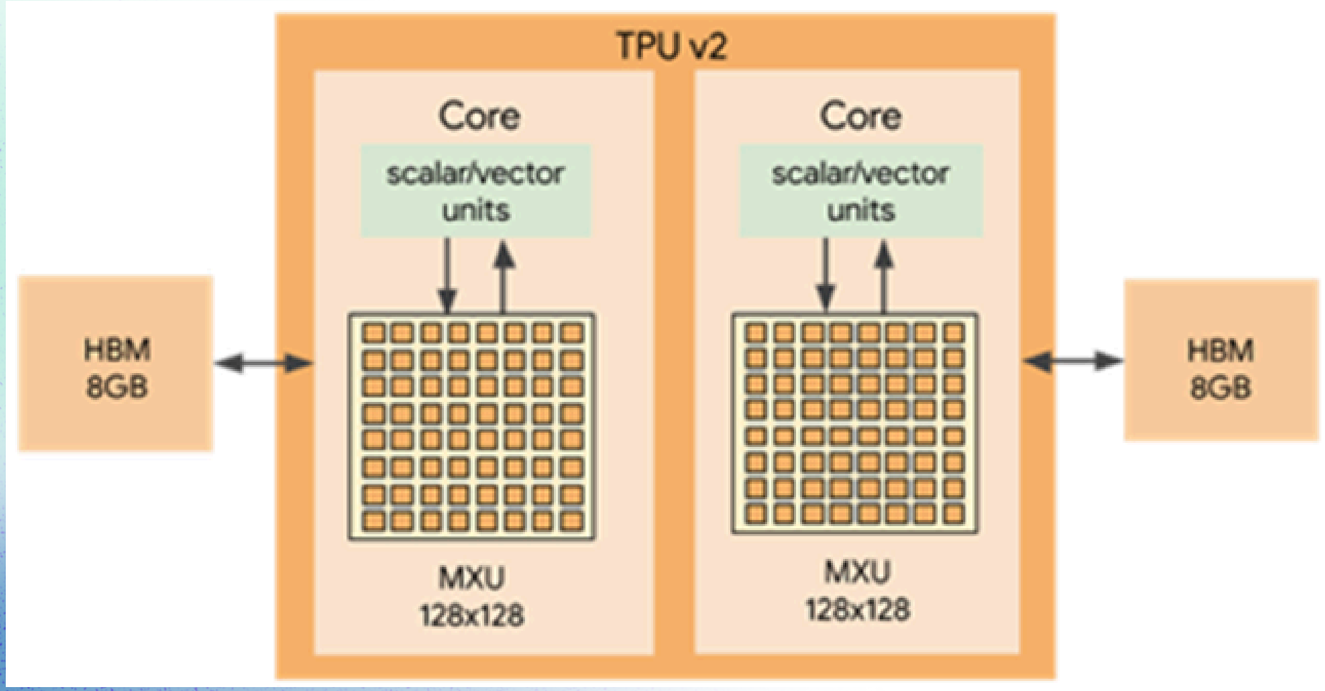
Monte Carlo Simulation with TensorFlow: Black-Scholes & LMM



Reporter: Jiayu Ban 476765
Danfeng Zhang 475114
Rana Ibrahimli
Yue Wu 474237
Supervisor: Dr. Hardy Wojciech

Overview

1 Introduction



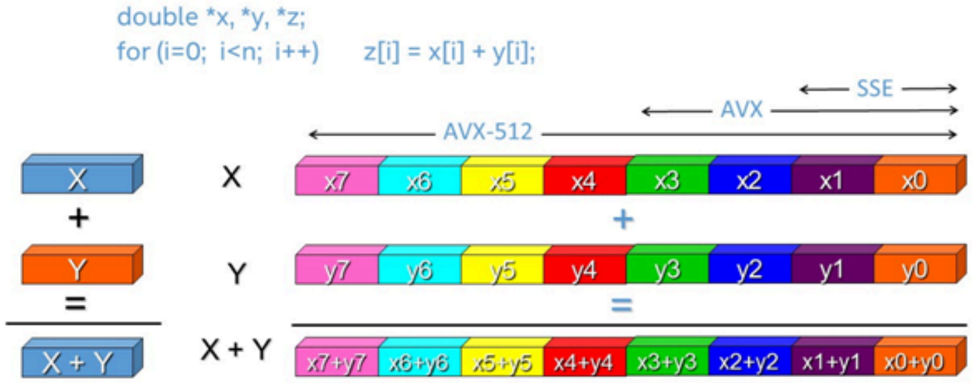
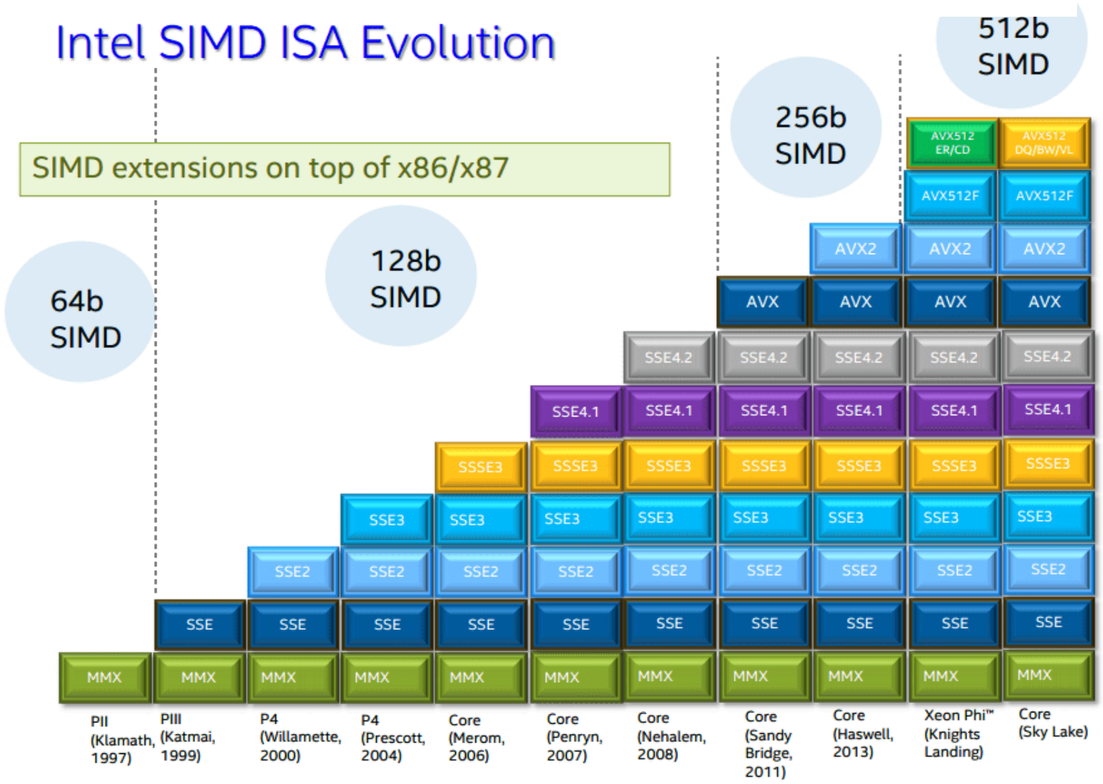
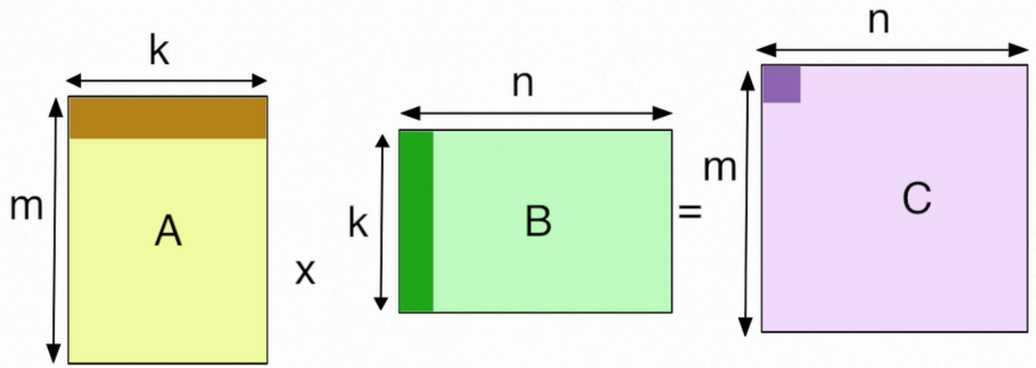


Figure 1 Scalar and vectorized loop versions with Intel® SSE, AVX and AVX-512.

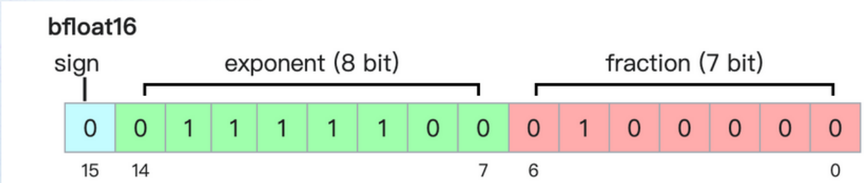


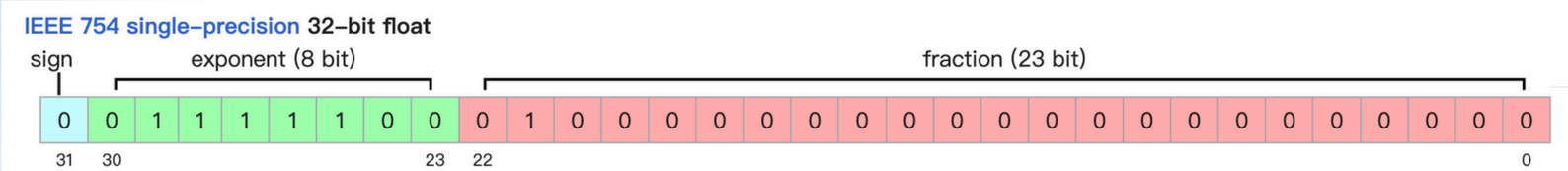
1 Introduction

GEMM as heart of finance algorithm

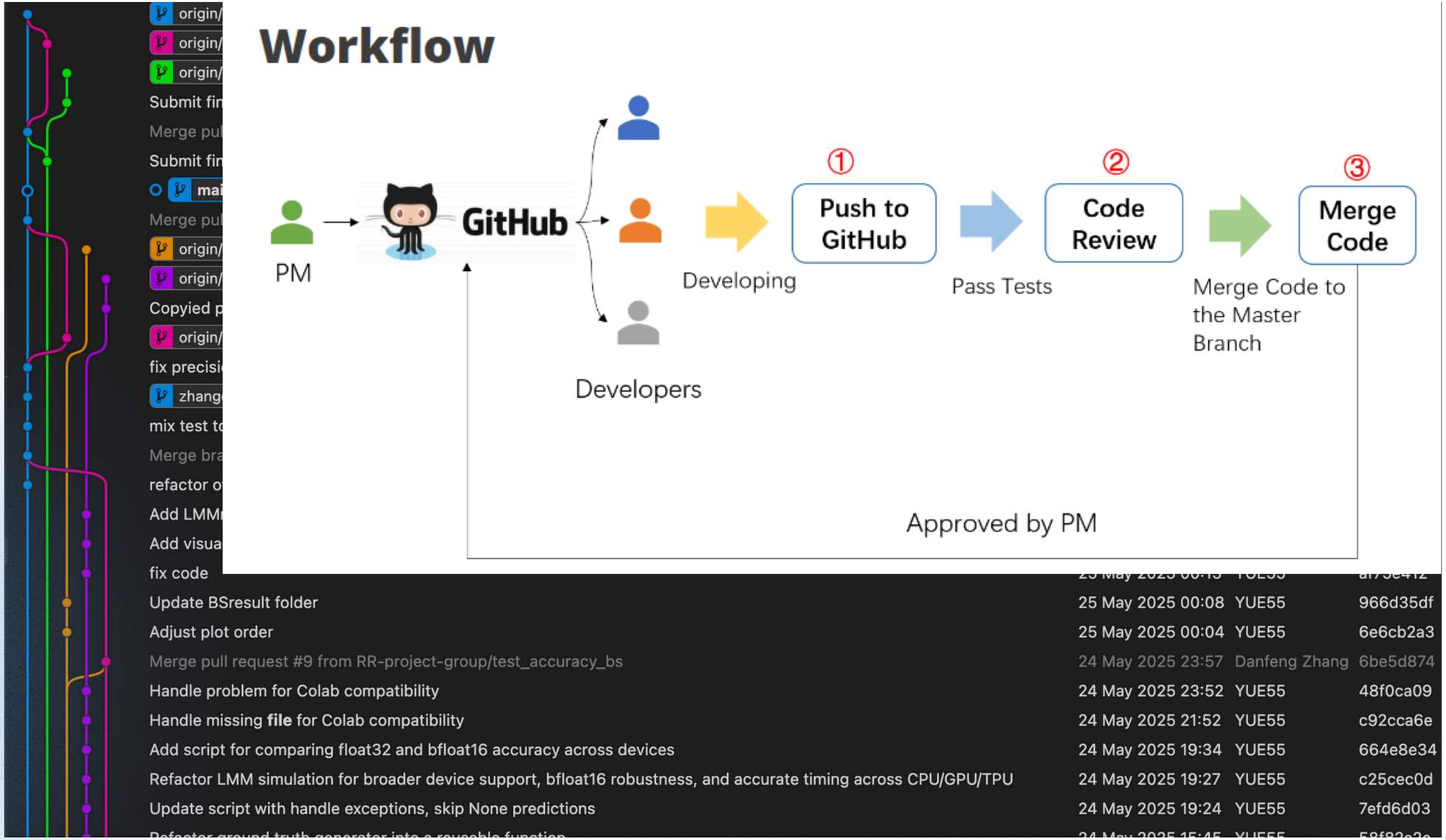


1000x1000x1000x2=2GFlops

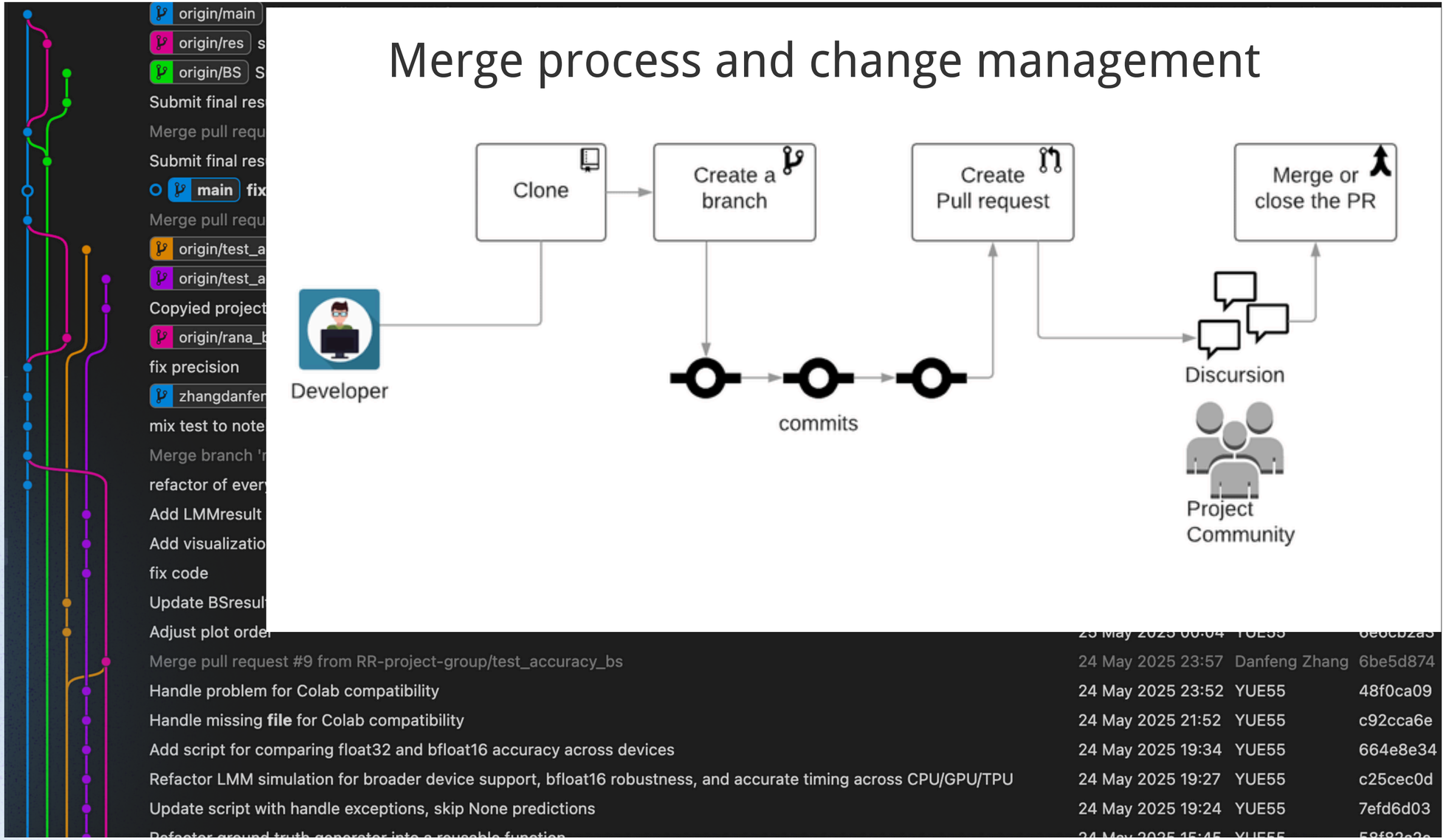




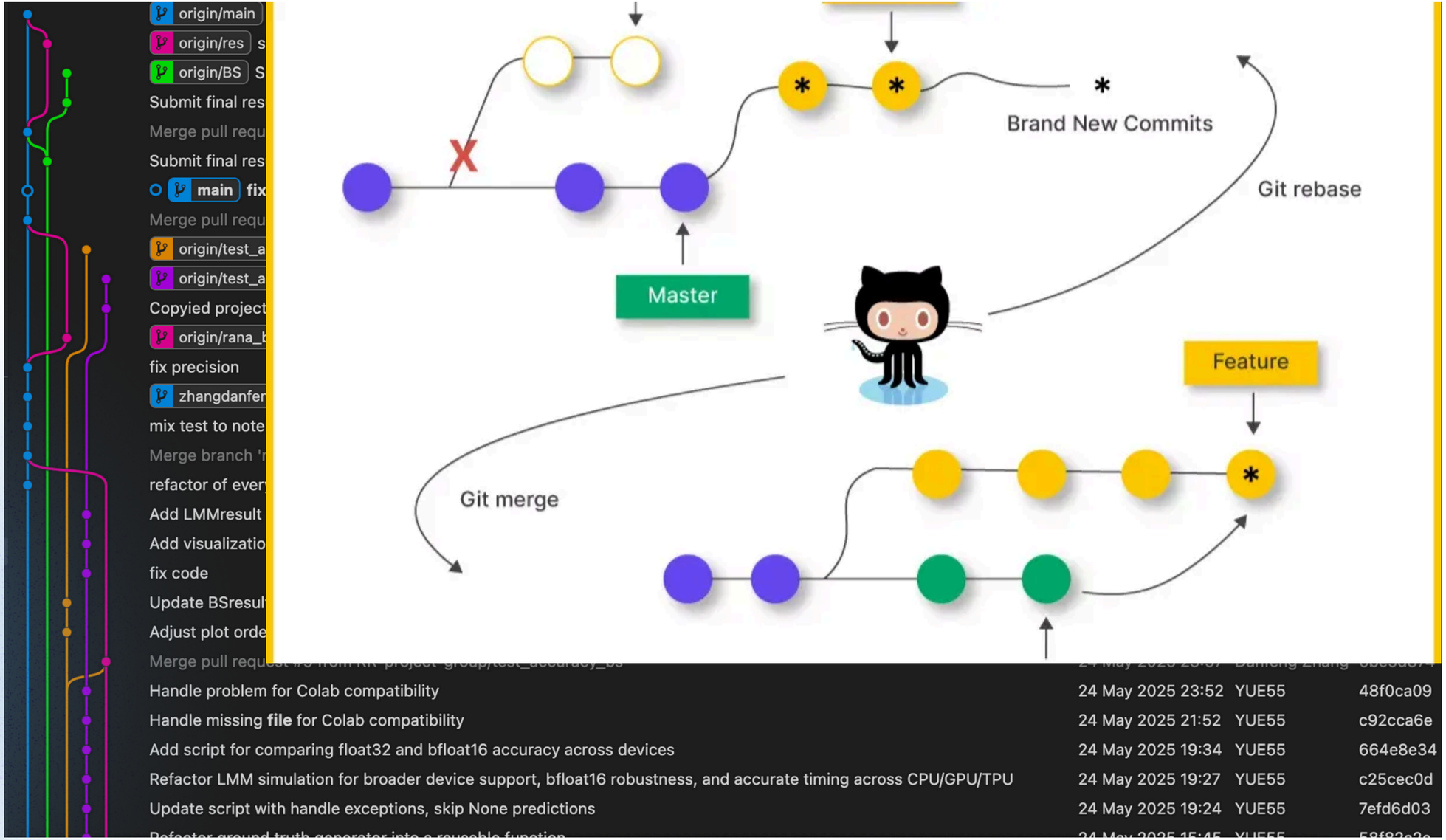
2 Workflow Overview



2 Workflow Overview



2 Workflow Overview



3 Implementation of the Two Algorithms

Black-Scholes option pricing with Monte Carlo simulation

```python
# ===== Monte Carlo simulation =====
@tf.function
def monte_carlo_bs_tpu(S0, K, T, r, sigma, n_paths, n_steps, dtype=tf.float32, seed=(0,0)):
    dt = T / n_steps
    # generate random numbers
    z = tf.random.stateless_normal(shape=(n_paths, n_steps), seed=seed, dtype=dtype)
    # generate log returns
    log_returns = ((r - 0.5 * sigma**2) * dt + sigma * tf.sqrt(dt) * z)
    log_paths = tf.cumsum(log_returns, axis=1)
    ST = S0 * tf.exp(log_paths[:, -1])  # final price
    payoff = tf.nn.relu(ST - K)
    price = tf.exp(-r * T) * tf.reduce_mean(payoff)
    return price

def run_bs_simulation(n_paths, n_steps, dtype=tf.float32, seed=(42, 42)):
    S0 = tf.constant(100.0, dtype=dtype)
    K = tf.constant(120.0, dtype=dtype)
    T = tf.constant(1.0, dtype=dtype)
    r = tf.constant(0.05, dtype=dtype)
    sigma = tf.constant(0.2, dtype=dtype)

    return tf.cast(monte_carlo_bs_tpu(S0, K, T, r, sigma, n_paths, n_steps, dtype=dtype, seed=seed), tf.float32)
```

- Parameter:
  S0: 100
  K: 120
  T: 1
  r: 0.05
  σ: 0.2
- Equation:

$$\Delta \ln S = \left(r - \frac{1}{2}\sigma^2\right) dt + \sigma\sqrt{dt}Z$$

Option Price = e^(-r×T) × Expected payoff

- Highlight:
  @tf.function

3 Implementation of the Two Algorithms

Libor Market Model path simulation with Cholesky factorization

```python
def generate_correlation_matrix(N, rho=0.5):
    corr = np.zeros((N, N), dtype=np.float32)
    for i in range(N):
        for j in range(N):
            corr[i, j] = rho ** abs(i - j)
    return corr

def sample_normals(shape, dtype):
    samples = tf.random.normal(shape, dtype=tf.float32)
    return tf.cast(samples, dtype)

@tf.function
def simulate_lmm_paths(cov, paths, T, N, dtype):
    cov_float32 = tf.cast(cov, tf.float32)
    chol = tf.linalg.cholesky(cov_float32)
    if dtype == tf.bfloat16:
        chol = tf.cast(chol, tf.bfloat16)

    normals = sample_normals((paths, T, N), dtype)
    libor_t = tf.ones((paths, N), dtype=dtype)
    paths_arr = []
    for t in range(T):
        z = tf.matmul(normals[:, t, :], chol, transpose_b=True)
        libor_t = libor_t * tf.exp(-0.5 * 0.01 + 0.01 * z)
        paths_arr.append(libor_t)
    return tf.stack(paths_arr, axis=1)
```

- Equation

$$\mathrm{corr}_{i,j} = \rho^{|i-j|} \qquad \rho = 0.5$$

$$L_{t+1} = L_t \cdot \exp\left(-\frac{\sigma^2}{2}\Delta t + \sigma\sqrt{\Delta t} \cdot z_t\right)$$

- Highlight
  mixed precision

6.1.2 Multi-variate process simulation to benchmark TPUs' MXU: Multi-variate stochastic simulations represent heavier computational workloads than their uni-variate counterparts whenever they involved correlated dynamics. A matrix/matrix multiply is then involve at each step of the simulation when computing the product of the volatility matrix with the multidimensional normally distributed stacked PRNs. In such a setting, the speed of the MXU on TPUs may be beneficial, but, as it uses a custom floating point representation, one needs to assess that no substantial numerical precision bias appears.

Basket European option: We price an at-the-money Basket European call with 2048 underlyings whose price is initially 100. The interest rate is 0.05 and the volatility matrix we use is a historical estimate based on market data collected on daily variations of randomly selected stocks from the Russell 3000 through 2018. 2K samples are used for each of the 100 simulations. Simulations now involve matrix multiplications corresponding to Equation (1.1) and therefore the MXU of the TPU is used with a reduced bfloat16 precision. All other computations on TPU run in single precision. In Figure 4 we present the estimates provided in mixed precision on TPU and compare them with single and double precision estimates. We find that running simulations on TPU does not introduce any significant bias while offering substantial speed ups compared to GPUs.

4 Accuracy Tests and Analysis

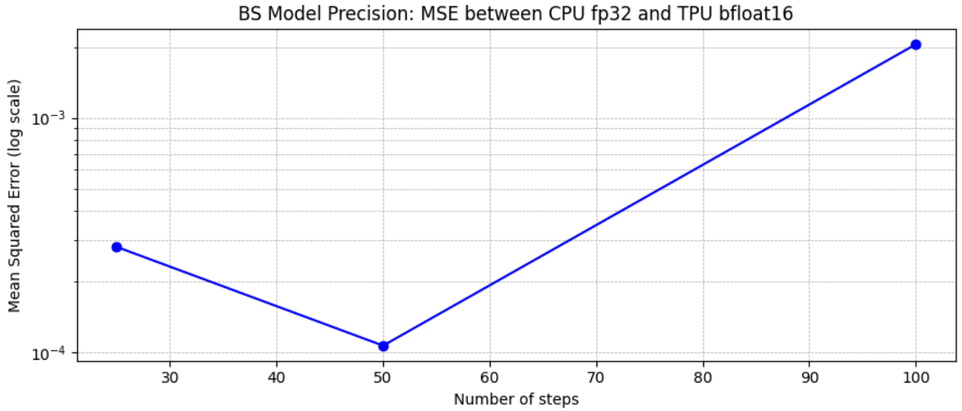
- Test Setup:
  a. Baseline: CPU (float32 precision) as ground truth
  b. Comparison Target: TPU (bfloat16 precision)
- Accuracy Metrics: Mean Squared Error (MSE)

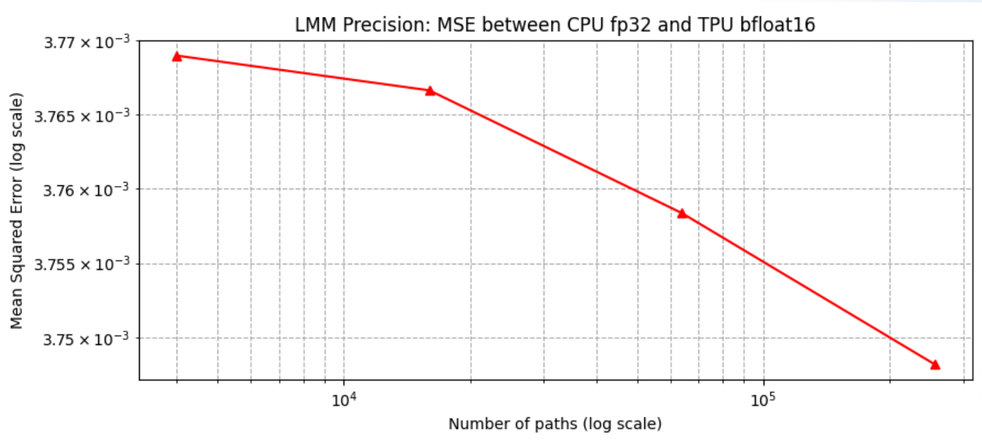
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

- Variable Parameters Applied:
  a. Black-Scholes(BS) Model:
     Varying time steps: n_steps = [25, 50, 100]

  b. Libor Market Model(LMM):
     Varying Monte Carlo paths: n_paths = [4000, 16000, 64000, 256000]

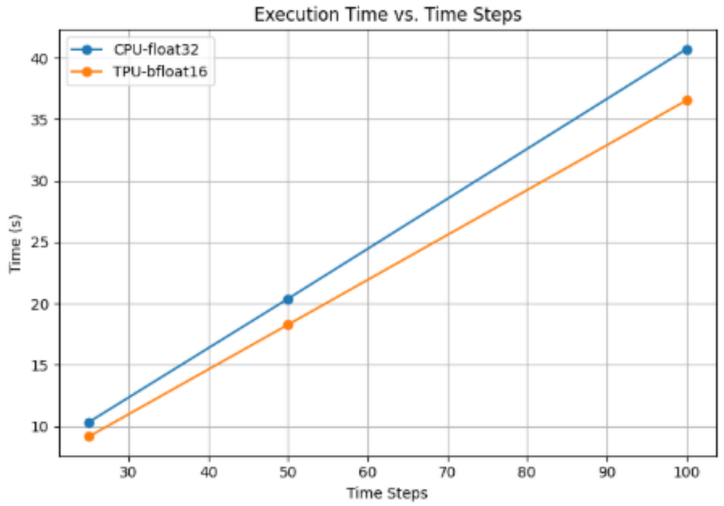
- Result Analysis: MSE stays around 10^-3, which indicates that despite reduced precision, TPU bfloat16 delivers results close to CPU float32 under the both models.





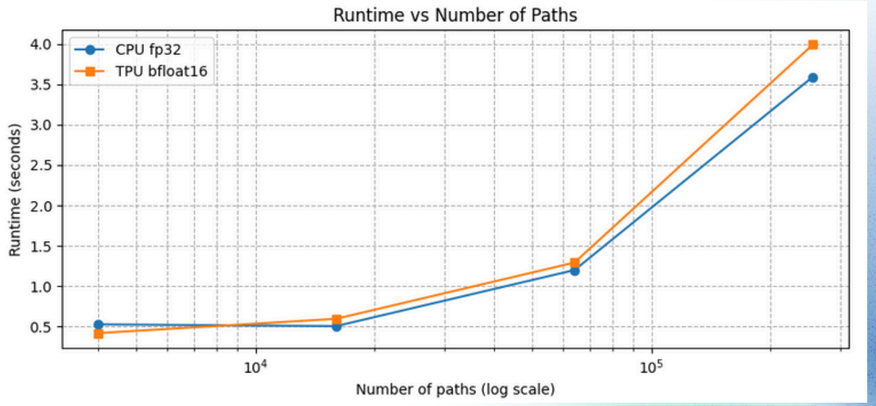
5 Runtime Performance and Analysis

- Benchmarked BS and LMM simulations using float32 (CPU); TPU fallback was not available.
- BS simulation tested with time steps: 25, 50, 100 → consistent option prices.
- LMM simulation tested with paths: 4,000 to 256,000 → runtime scaled smoothly.
- Graphs confirm minor accuracy tradeoff with bfloat16, acceptable for finance.
- All tests run on CPU — performance gain expected on real TPU.



📊 Benchmark Results:

	label	dtype	n_steps	price
0	CPU-float32	float32	25	3.248831
1	CPU-float32	float32	50	3.244730
2	CPU-float32	float32	100	3.248410



6 Conclusion

1. TPUs enable a responsive and interactive experience with a speed higher than or comparable to that of CPUs running TensorFlow.

2. TPUs are indeed accurate enough, fast and easy to use for financial simulation.

Reference

- https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.2
- https://docs.pytorch.org/docs/stable/torch.html
- https://en.wikipedia.org/wiki/Monte_Carlo_method
- https://en.wikipedia.org/wiki/Tensor_Processing_Unit
- https://en.wikipedia.org/wiki/Central_processing_unit
- https://en.wikipedia.org/wiki/Graphics_processing_unit
- https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model
- https://en.wikipedia.org/wiki/LIBOR_market_model
- https://en.wikipedia.org/wiki/Mean_squared_error
- https://en.wikipedia.org/wiki/Mean_absolute_error
- https://haibal.com/documentation/metric-mean-relative-error/#:~:text=The%20Mean%20Relative%20Error%20is,normalizer%2C%20producing%20a%20relative%20error.
- https://en.wikipedia.org/wiki/Single-precision_floating-point_format
- https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

THANK YOU

FOR YOUR ATTENTION!

ANY QUESTIONS