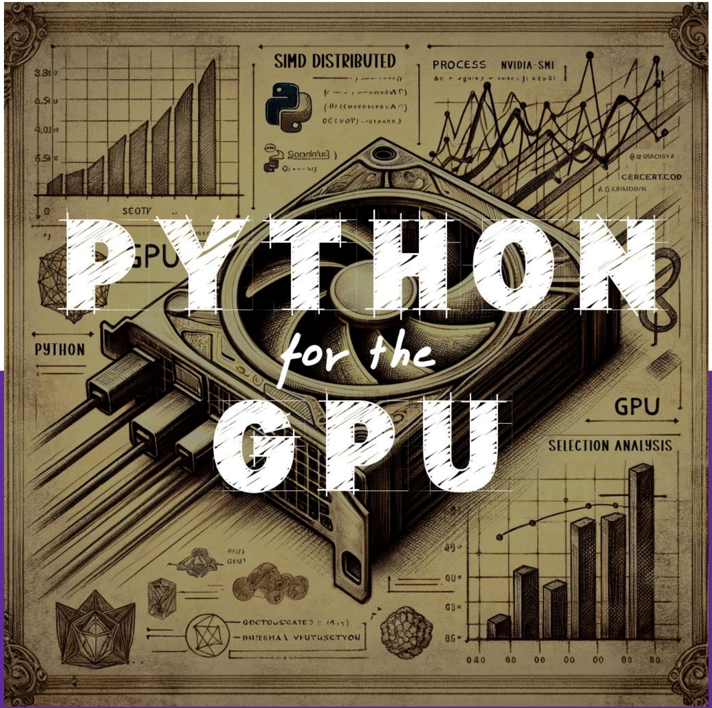

PYTHON
for the
GPU
SIMD DISTRIBUTED
PROCESS NVIDIA-SMI
GPU
PYTHON
GPU
SELECTION ANALYSIS

Outline

- Foundational differences between CPUs and GPUs
- General layout of GPUs
- How to use Numba to write kernels for a GPU
- Monte Carlo simulations in finance

CPU v GPU

	CPU	GPU
Design	Low Latency	High Throughput
Optimized For	Single Thread	Many Threads
Memory Bandwidth	Lower	Higher
Cache Size	Larger	Smaller

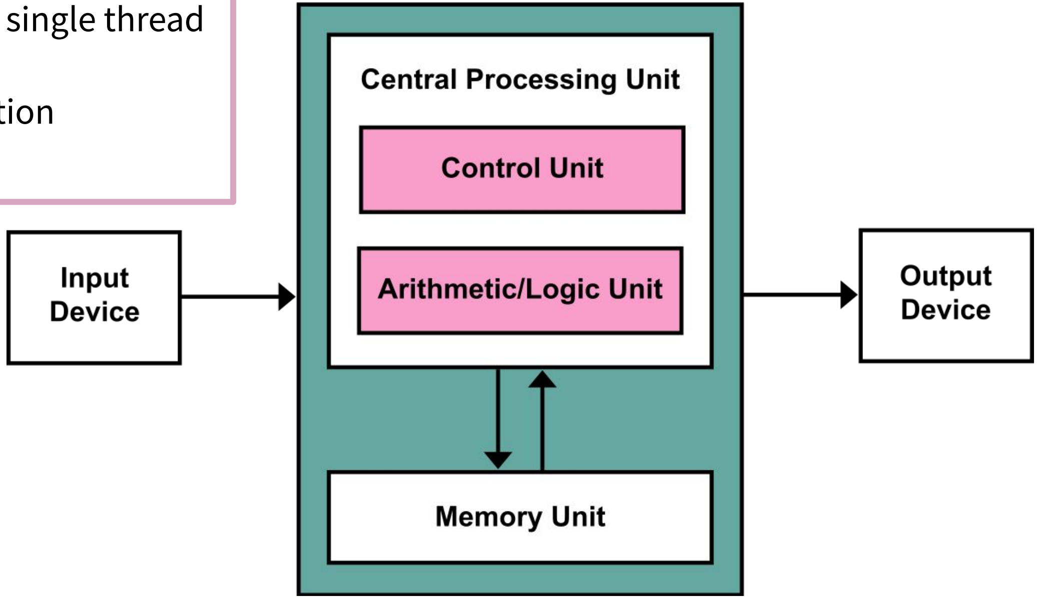
CPU = Central Processing Unit

Low-latency oriented design

- Optimizes for *sequential* performance of a single thread
- Large cache memory for data access
- Arithmetic units increase power consumption
- Complex instruction sets in control units

Cons

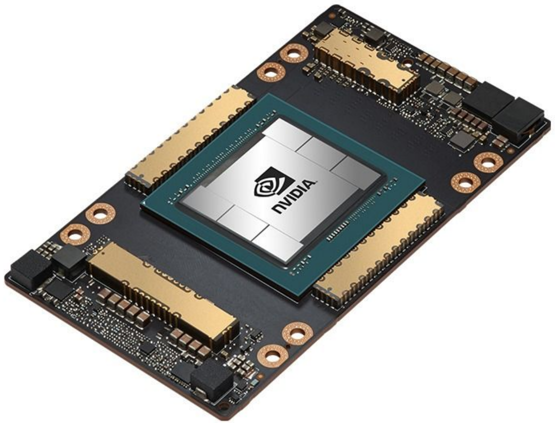
- Lower memory bandwidth
- More power to the ALU
- More space for memory



GPU = Graphics Processing Unit

High Throughput design

- Optimizes for parallel performance of many threads
- Generally smaller cache, registers and memory sizes
- Extremely high memory bandwidth
- Simpler instruction sets



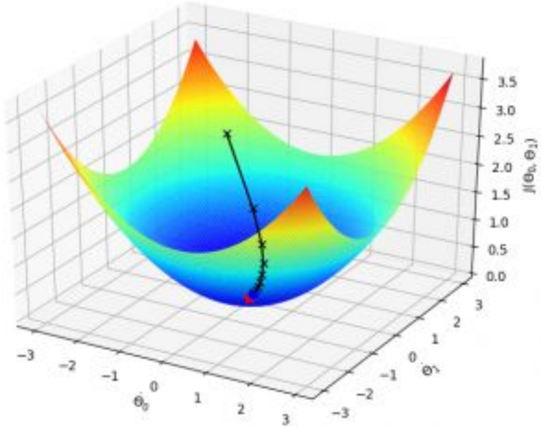
Cons

- Slower thread execution speed
- More power to memory bandwidth means each worker is weaker
- Many threads mean less space for cache and registers

GPGPU = General Purpose Graphics Processing Unit

Many scientific calculations are **embarrassingly parallel**

- Gradient Descent in Machine Learning
- Fourier Transforms
- Large matrix operations
- *Monte Carlo Simulations*



In 2007, Nvidia introduces both

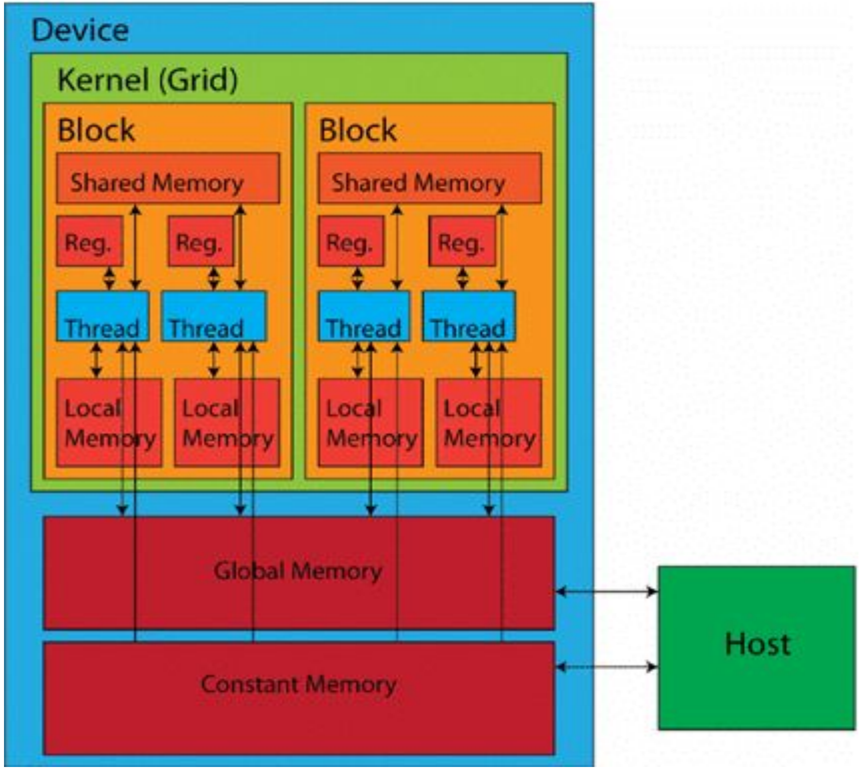
- Tesla GPGPU, which allows for user specific data structures
- Compute Unified Device Architecture (CUDA), which exposes GPU internals

Software and Hardware Issues

- CUDA is a Nvidia specific software
- OpenCL (Open Computing Language, 2009) runs on Nvidia and AMD chips
- Nvidia chips are generally faster than AMD's chips
- CUDA is faster than OpenCL on Nvidia chips

Higher level libraries can make you hardware and software independent!

GPU Hardware



Threads	Single unit of execution
Local Memory	R/W Memory for single thread
Blocks	Collection of threads
Shared Memory	R/W memory accessible by block
Grid	Layout of blocks (1-3 dimensions)
Global Memory	R/W memory globally accessible
Constant Memory	Small space for read-only access
Kernel	Function executing on grid
Host	CPU process launching kernel

Numba

- Numba is JIT compiler for both CPUs and GPUs
- Compiles a subset of Python and NumPy for the GPU
- For both Nvidia and AMD chips

Numba

NVIDIA.

CUDA®

```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication
of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Alternatives to Numba: CuPy

- CuPy is NumPy for GPUs
- Extensive library of array operations
- Ability to add custom “raw” kernels
- For both Nvidia and AMD chips

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Alternatives to Numba: PyCuda

- Fastest library
- Requires raw kernels
- Most like CUDA syntax
- For Nvidia chips only

```python
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print(dest-a*b)
```

Declaring a kernel

- A **kernel** is the main function launched on the GPU
- Each thread of the GPU executes the kernel

```
from numba import cuda
import numpy as np

@cuda.jit
def kernel(A, B, C):
    """A + B = C"""
    # Get index of thread launched in a 2 dimensional grid
    ii, jj = cuda.grid(2)

    # check if thread index corresponds to indices of matrices
    if (ii < A.shape[0]) and (jj < A.shape[1]):
        # C_ij = A_ij + B_ij
        C[ii, jj] = A[ii, jj] + B[ii, jj]
```

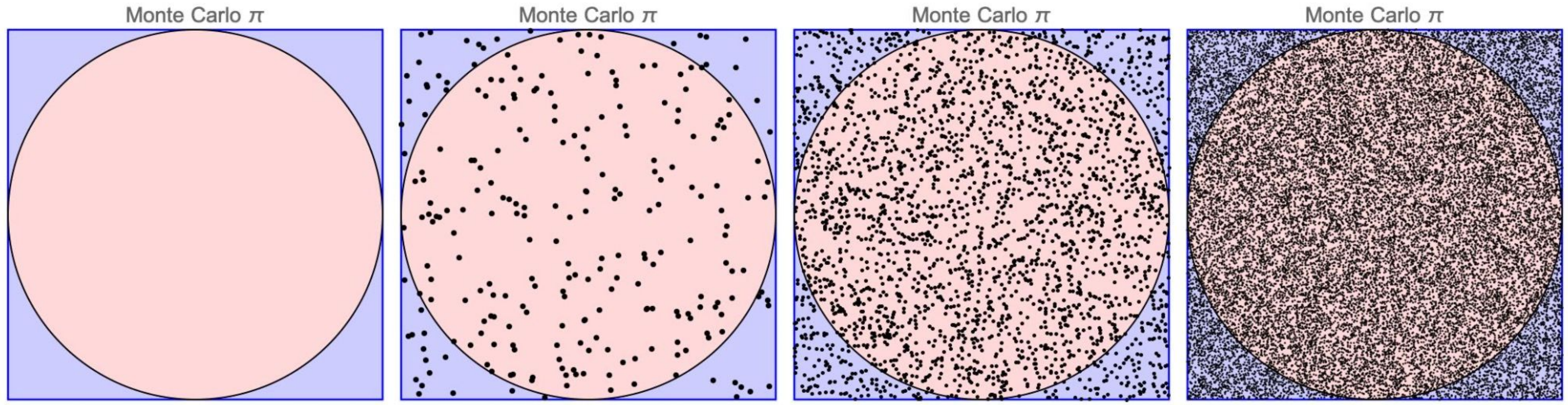
Declaring threads and blocks

- Threads are organized into blocks and assigned to processors (max of 1024 threads/block).
- Blocks are organized into a grid (1, 2 or 3 dimensions).

```
N = 10240
A = np.random.randn(N, N).astype(np.float32)
B = np.random.randn(N, N).astype(np.float32)
C = np.zeros((N, N), dtype=np.float32)

threads_per_block = (16, 16)
number_of_blocks = (math.ceil(N / 16), math.ceil(N / 16))

# A, B, C are moved into global memory
# number_of_blocks * threads_per_block launched
kernel[number_of_blocks, threads_per_block](A, B, C)

(C == (A + B)).all() # = True
```

Monte Carlo simulations

Monte Carlo is a **computational algorithm** that uses repeated **random sampling** to calculate an **expected value.**



Betting on coin tosses

Consider a bet on a coin toss

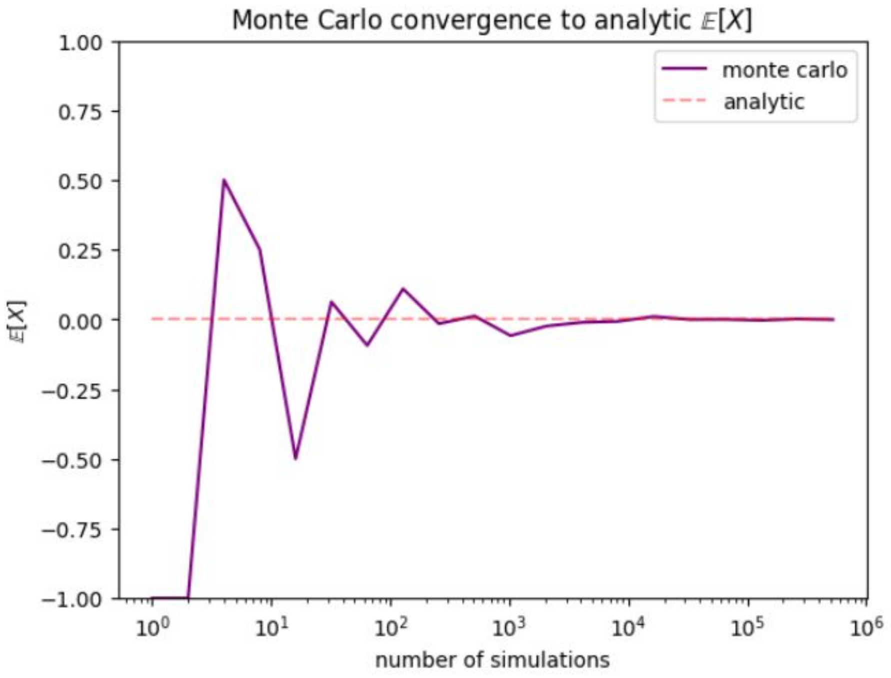
$$X = \begin{cases} +1 & P(X=1) = \frac{1}{2} \\ -1 & P(X=-1) = \frac{1}{2} \end{cases}$$

Here, we have an *analytic* formula for calculating the expected value

$$\begin{aligned} \text{Expected Value} = \mathbb{E}[X] &= P(X=1) \times 1 + P(X=-1) \times -1 \\ &= \frac{1}{2} * 1 + \frac{1}{2} * -1 \\ &= 0 \end{aligned}$$

Expected value via Monte Carlo

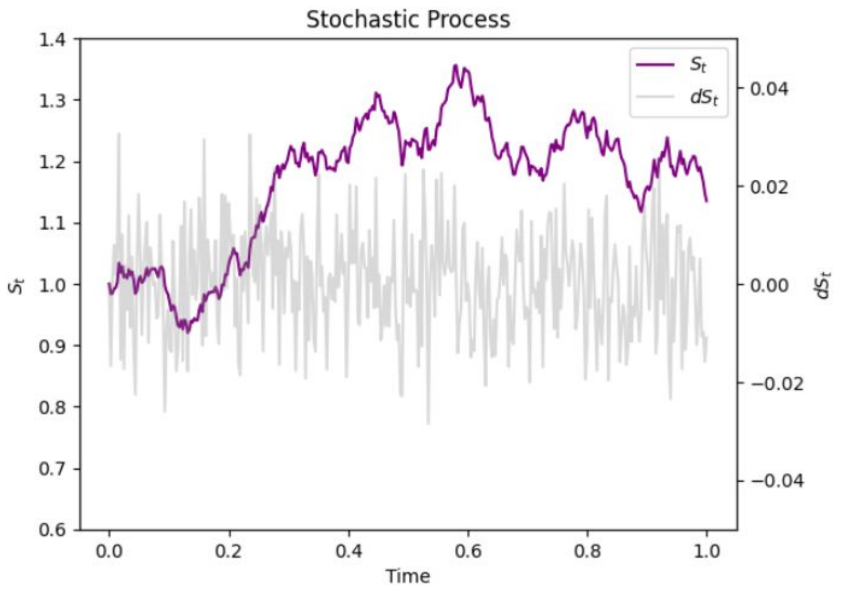
```python
import numpy as np
import matplotlib.pyplot as plt

# set random number generator
rng = np.random.default_rng(1234)


def expected_value_coin_toss(nsims: int) -> float:
    # randomly choose heads (1) or tails (-1)
    coin_tosses = rng.choice([1.0, -1.0], size=(nsims,))

    # expected value is average across trials
    expected_value = coin_tosses.mean()

    return expected_value


nsims = 2 ** np.arange(20)
epvs = [expected_value_coin_toss(nsim) for nsim in nsims]
```



Monte Carlo methods in finance

$$S_t = S_{t-1} * \exp(dS_t)$$
$$dS_t = (r - \sigma^2/2) \times dt + \sigma \times \sqrt{dt} \times dWt$$

Variable	Definition
dt	time step (e.g. 1 day)
r	growth (in prportion to time step)
σ	volatility
dW_t	random variable at time t (from normal distribution)
dS_t	change over dt
S_t	level at time t



Asian Option

= max(average(stock price) - strike price, 0.0)

The value at time T along a single path is

$$\text{Value}_T = \left(\frac{1}{T}\int_0^T S_t dt - K\right)^+$$

The value at time 0 along a single path is

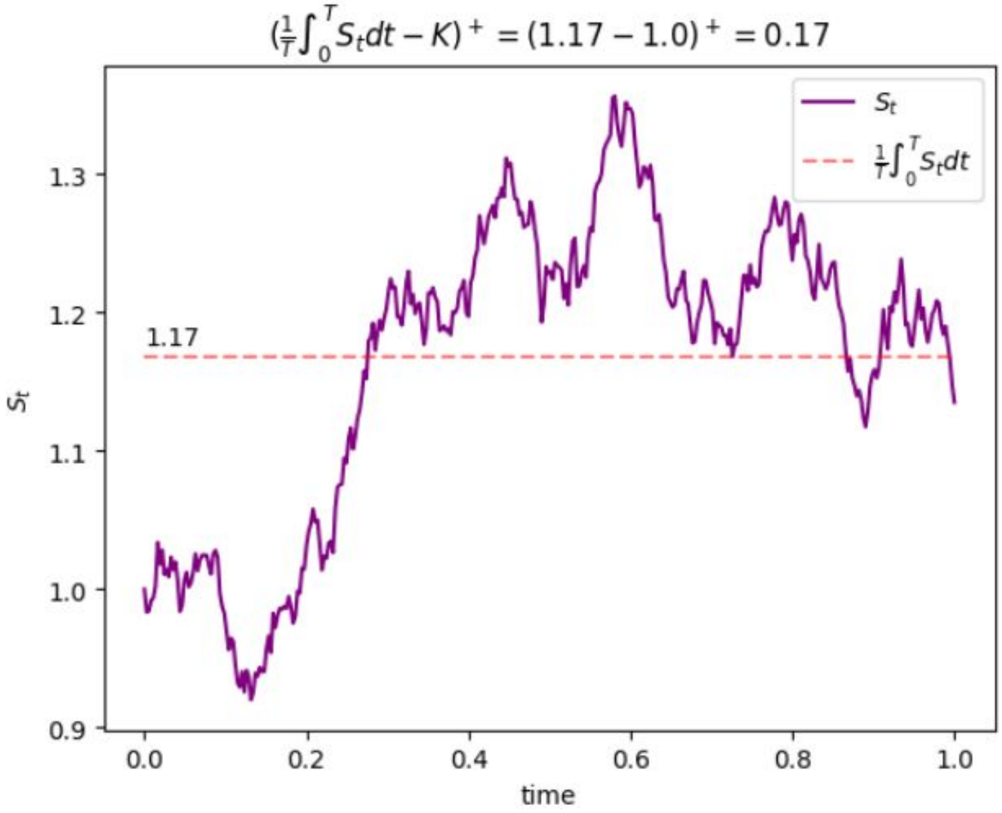
$$\begin{aligned}\text{Value}_0 &= e^{-rT} \times \text{Value}_T \\ &= e^{-rT}\left(\frac{1}{T}\int_0^T S_t dt - K\right)^+\end{aligned}$$

The expected value at time 0 is

$$\begin{aligned}\mathbb{E}[\text{Value}_0] &= \mathbb{E}[e^{-rT} \times \text{Value}_T] \\ &= \mathbb{E}[e^{-rT}\left(\frac{1}{T}\int_0^T S_t dt - K\right)^+)\end{aligned}$$

Variable	Definition
$(\text{something})^+$	$\max(\text{something}, 0.0)$
K	strike price
e^{-rT}	discount factor

Visualizing Asian Option payoff



Monte Carlo on the CPU & on the GPU

Other Concepts

- Optimize using local, shared and constant memory
- Thread coalescing and memory striding
- Multiple kernels
- Multiple devices
- Optimize thread and block sizing

Thank you!