



# Implementing Quantitative Use Cases with Accelerated Computing for Faster Time-to-Insight

Ioana Boier, Jeff Larkin, Manolis Papadakis, Emanuel Scoullos | Nvidia GTC Live Special Event/March 21, 2023



An abstract, high-contrast image featuring vibrant green, translucent, fiber-like structures that swirl and loop against a solid black background. The green elements have a glassy, refractive quality, with internal reflections and refractions visible. The overall composition is dynamic and organic, resembling a microscopic view of a material or a complex data visualization.

# Agenda

- Acceleration for large-scale problems in Quant Finance

---
- "Speed of light" exploration of option portfolios in ISO C++

---
- Truly rapid prototyping in Python with cuNumeric

---
- Large language models for language and beyond (time series prediction and synthetic data generation)

---
- References and links to hands-on materials




# Acceleration for Large-Scale Problems in Quant Finance

Not Your Grandma's GPUs

The goal of this session is to provide hands-on examples of how to leverage *modern acceleration capabilities* for:

- 1. Large scale portfolio analyses and simulations (HPC use case)
- 2. Training and inference with LLMs. (AI use case)

Grandma	You After GTC 2023
Wrote low-level CUDA code in C	Write standard C++ vn. 17 or higher
Maintained separate code bases for CPU / GPU	Maintain a single code base for CPU / GPU
Developed from scratch	Leverage Nvidia SDKs via NGC containers
Used slow Python	Rapid prototype using accelerated Python
Trained models on CPU or a single GPU	Learn about multi-GPU frameworks: <ul style="list-style-type: none"><li>• cuNumeric</li><li>• Megatron-LM</li></ul>
Didn't have ChatGPT 	Learn how to leverage Transformers: <ul style="list-style-type: none"><li>• Synthetic data generation</li><li>• Time series prediction</li></ul>

# Related GTC Sessions

Upcoming

- March 21, 5pm ET: [No More Porting: GPU Computing with Standard C++ and Fortran \[S51043\]](#)
- March 22, 12pm ET: [Fireside Chat with Ilya Sutskever and Jensen Huang: AI Today and Vision of the Future \[S52092\]](#)
- March 22, 4pm ET: [FP8 Training with Transformer Engine \[S51393\]](#)
- March 23, 9am ET: [Option Pricing Model Calibration using Deep Learning \[S51284\]](#)
- March 23, 9am ET: [Accelerating HPC applications with ISO C++ on Grace Hopper \[S51054\]](#)
- March 23, 11am ET: [cuNumeric and Legate: How to Create a Distributed GPU Accelerated Library \[S51789\]](#)
- March 24, 4pm ET: [Synthetic Tabular Data Generation Using Transformers \[DLIT52224\]](#)

# Useful Links

- NGC catalog: <https://catalog.ngc.nvidia.com/>
- Launchpad: <https://www.nvidia.com/en-us/launchpad/>
- ISO C++ Parallelism: <https://docs.nvidia.com/hpc-sdk/compiler/c++-parallel-algorithms/index.html>
- HPC SDK: <https://developer.nvidia.com/hpc-sdk>
- cuNumeric library: <https://developer.nvidia.com/cunumeric>
- NeMo LLM Service: <https://www.nvidia.com/en-us/gpu-cloud/nemo-llm-service/>
- Megatron-LM github: <https://github.com/NVIDIA/Megatron-LM>
- Synthetic Data Generation blog: <https://developer.nvidia.com/blog/generating-synthetic-data-with-transformers-a-solution-for-enterprise-data-challenges/>





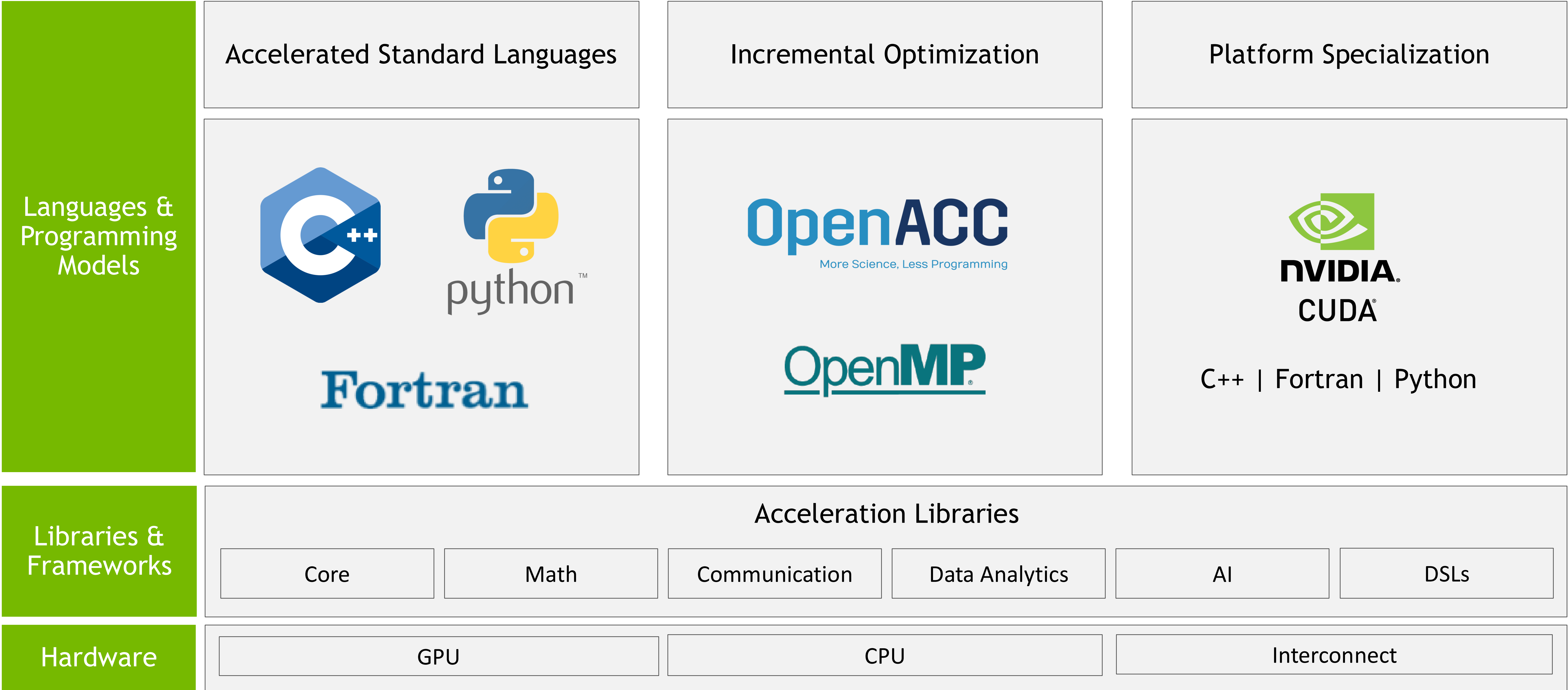
# "Speed of Light" Exploration of Option Portfolios with Standard C++

Jeff Larkin, Principal HPC Application Architect | Ioana Boier, Ph.D., Principal Solutions Architect |

Nvidia GTC Live Special Event/March 21, 2023



# Programming The NVIDIA Platform



# Accelerated Standard Languages

Parallel performance for wherever your code runs

## ISO C++

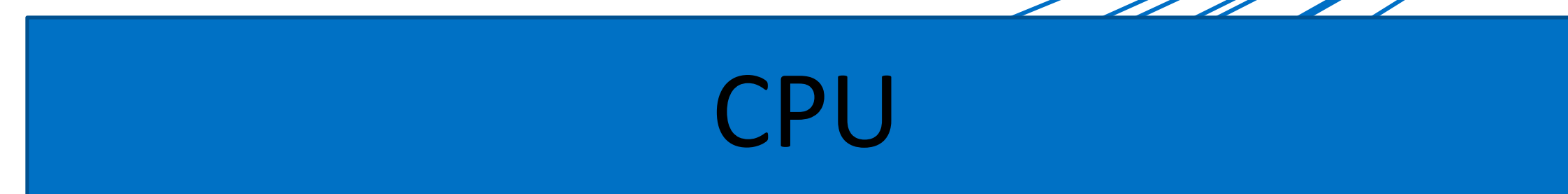
```
std::transform(par, x, x+n, y,  
              y, [=](float x, float y){  
                  return y + a*x;  
              })  
);
```

## ISO Fortran

```
do concurrent (i = 1:n)  
    y(i) = y(i) + a*x(i)  
enddo
```

## Python

```
import cunumeric as np  
...  
def saxpy(a, x, y):  
    y[:] += a*x
```



```
nvc++ -stdpar=multicore  
nvfortran -stdpar=multicore  
legate -cpus 16 saxpy.py
```

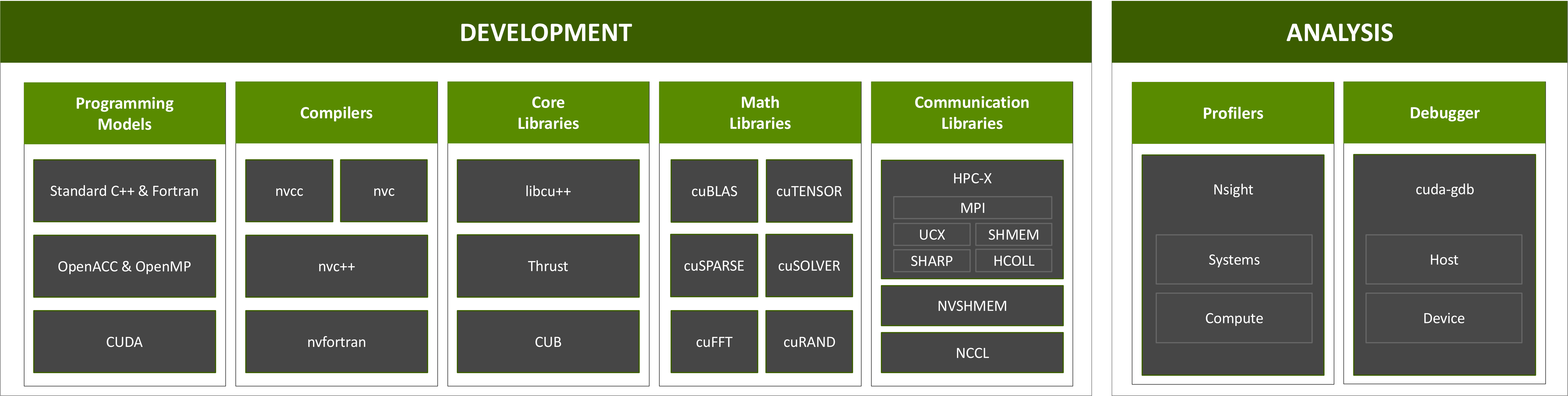


```
nvc++ -stdpar=gpu  
nvfortran -stdpar=gpu  
legate -gpus 1 saxpy.py
```



# NVIDIA HPC SDK

Available at [developer.nvidia.com/hpc-sdk](https://developer.nvidia.com/hpc-sdk), on NGC, via Spack, and in the Cloud



Develop for the NVIDIA Platform: GPU, CPU and Interconnect  
Libraries | Accelerated C++ and Fortran | Directives | CUDA  
x86\_64 | Arm | OpenPOWER  
7-8 Releases Per Year | Freely Available



# HPC PROGRAMMING IN ISO C++

ISO is the place for portable concurrency and parallelism

## C++17 & C++20

### Parallel Algorithms

- Parallel and vector concurrency

### Forward Progress Guarantees

- Extend the C++ execution model for accelerators

### Memory Model Clarifications

- Extend the C++ memory model for accelerators

### Ranges

- Simplifies iterating over a range of values

### Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators

## Preview support coming to NVC++

## C++23

### `std::mdspan`

- HPC-oriented multi-dimensional array abstractions.
- [Preview Available Now](#)

### Range-Based Parallel Algorithms

- Improved multi-dimensional loops

### Extended Floating Point Types

- First-class support for formats new and old: `std::float16_t/float64_t`

## And Beyond

### Senders/Receivers

- Standardized mechanism for asynchrony in the C++ standard library
- Simplify launching and managing parallel work across CPUs and accelerators
- [Preview Available Now](#)

### Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries
- [Preview Available Now](#)

### MDArray and SubMDSpan

- Expands the capabilities of C++23 MDSpan
- [Preview Available Now](#)



# Options Universe Exploration

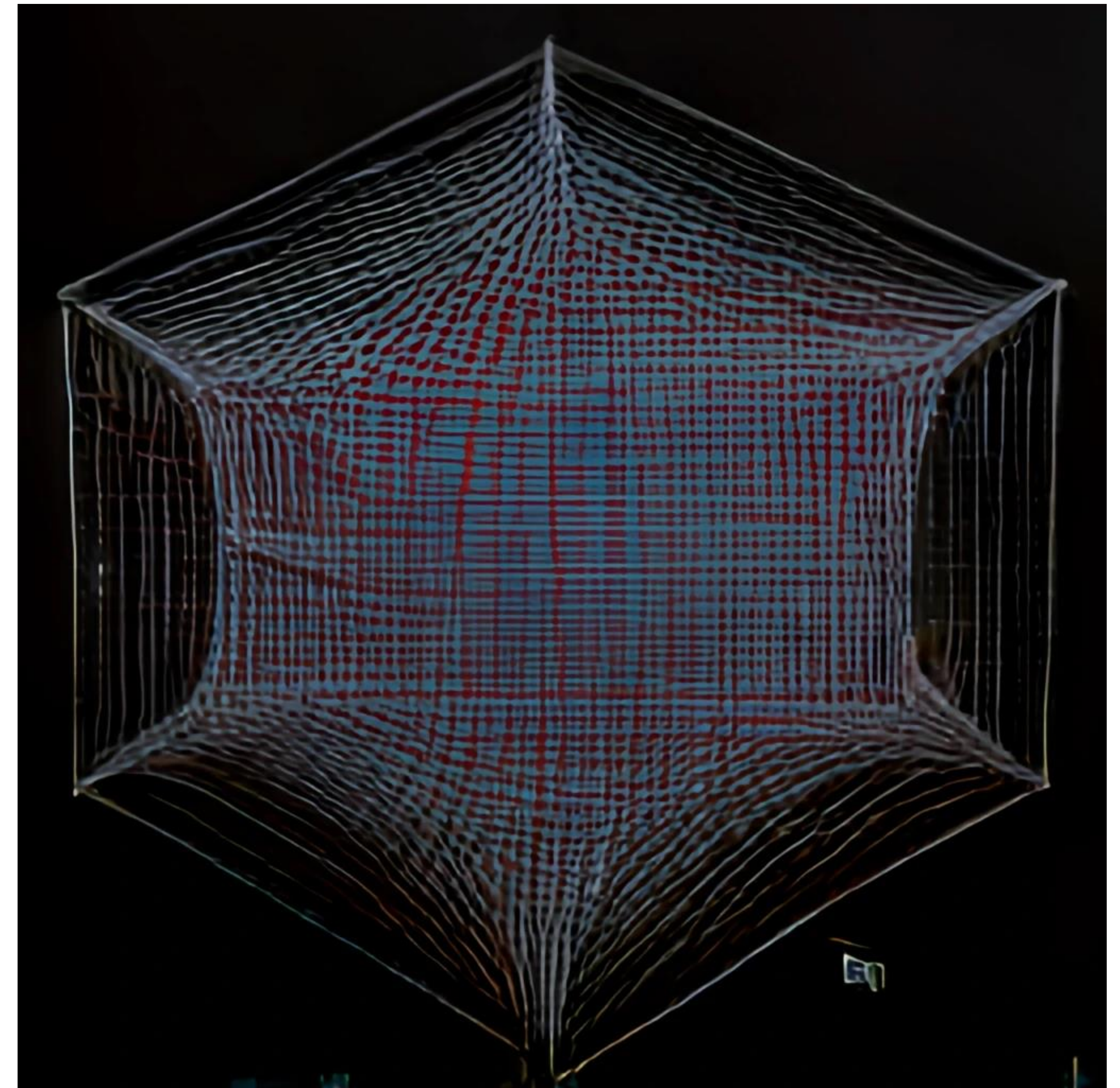
## Use Cases

### Large option portfolios:

- Many options
- Several asset classes
- Portfolio dynamics (backtesting / simulation)

### Strategy exploration:

- Hypothetical scenarios with full P&L distribution estimates
- Risk management of non-linear portfolios
- Portfolio optimization



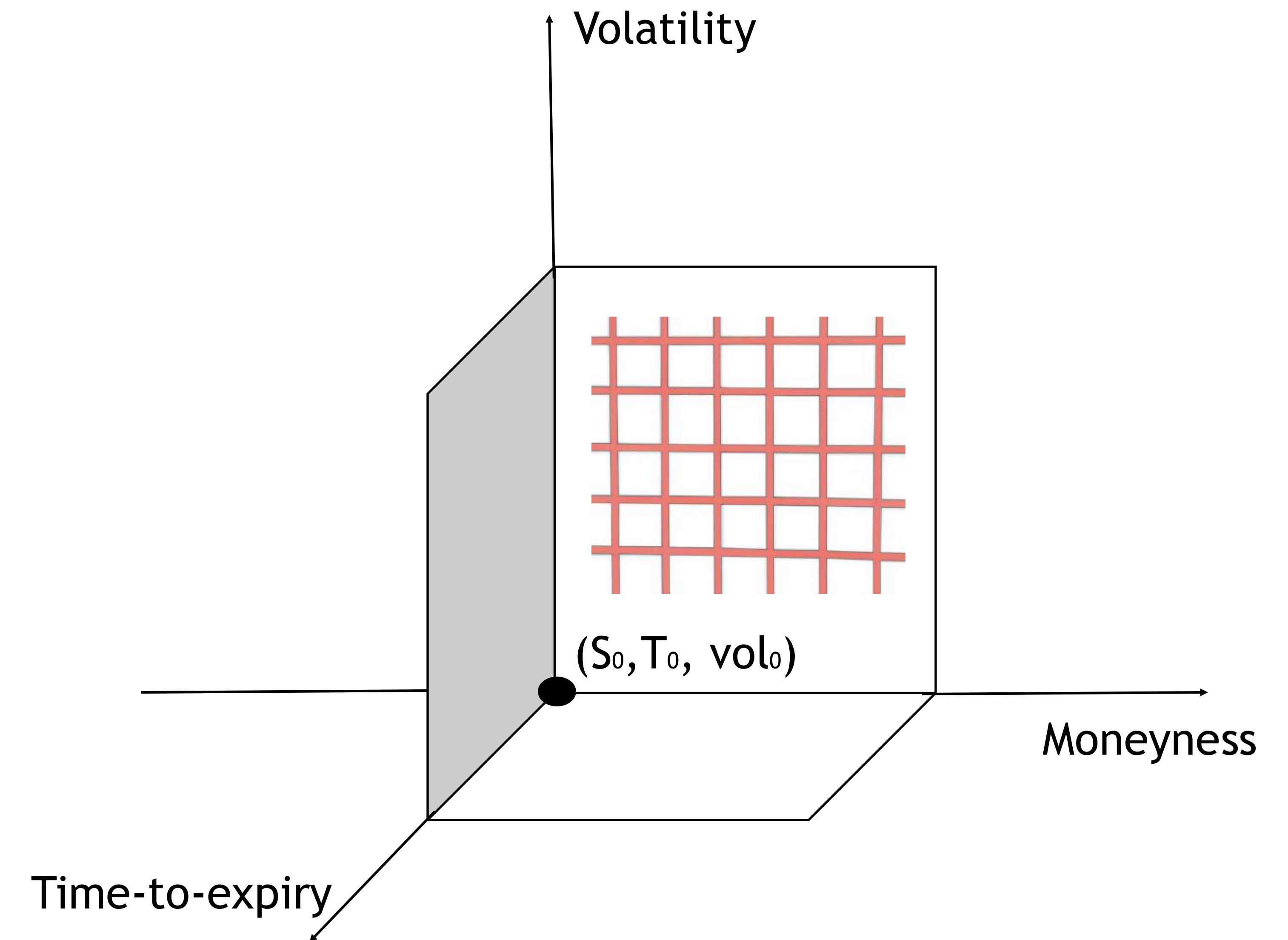
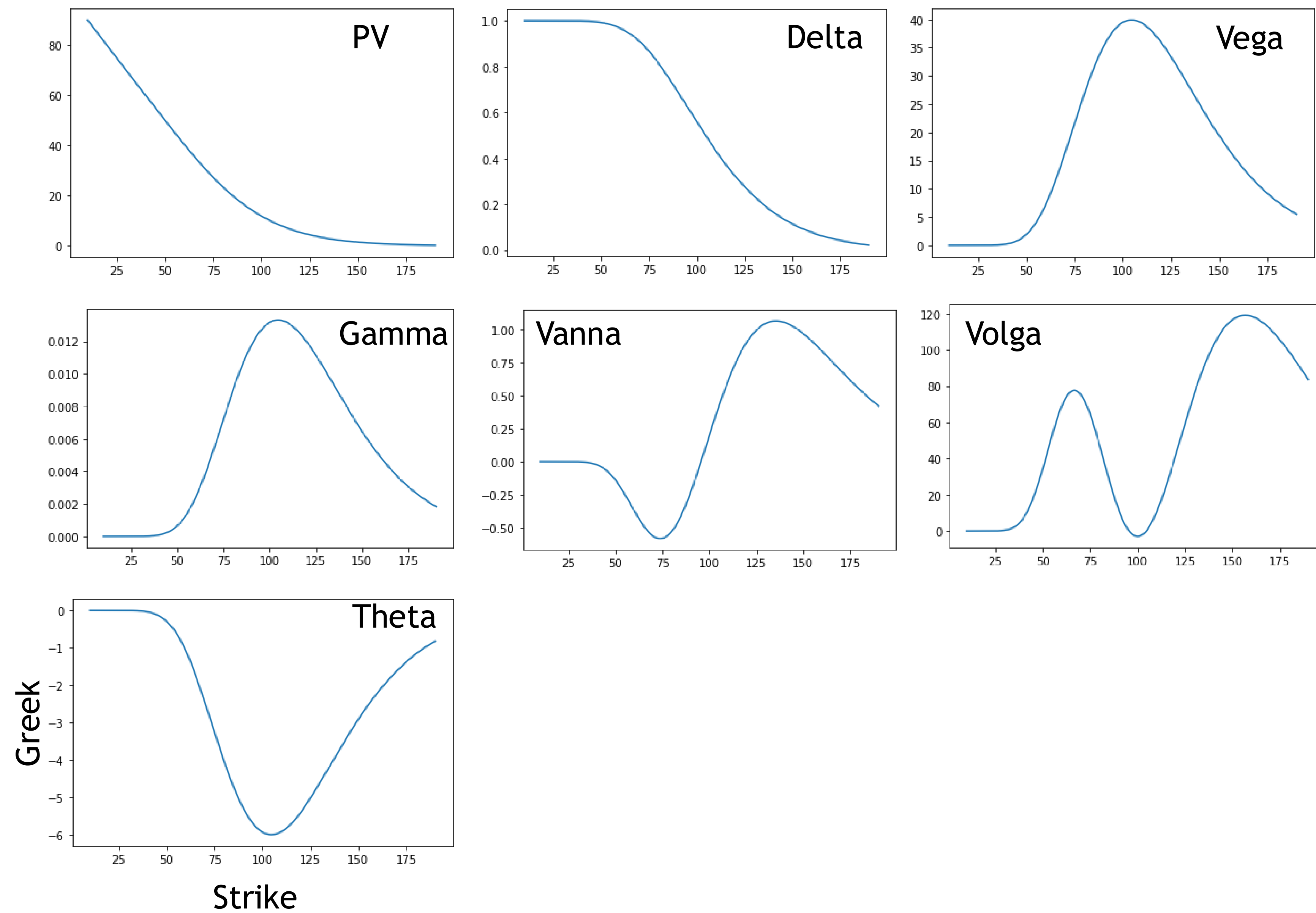


# Example #1

## Massively Parallel

### Fast PV & Greeks:

- Black-Scholes-Merton model
- 15M+ options (calls and puts)
- Large grid: time-to-maturity, volatility, and moneyness
- Greeks: premium, delta, vega, theta, gamma, vanna, volga





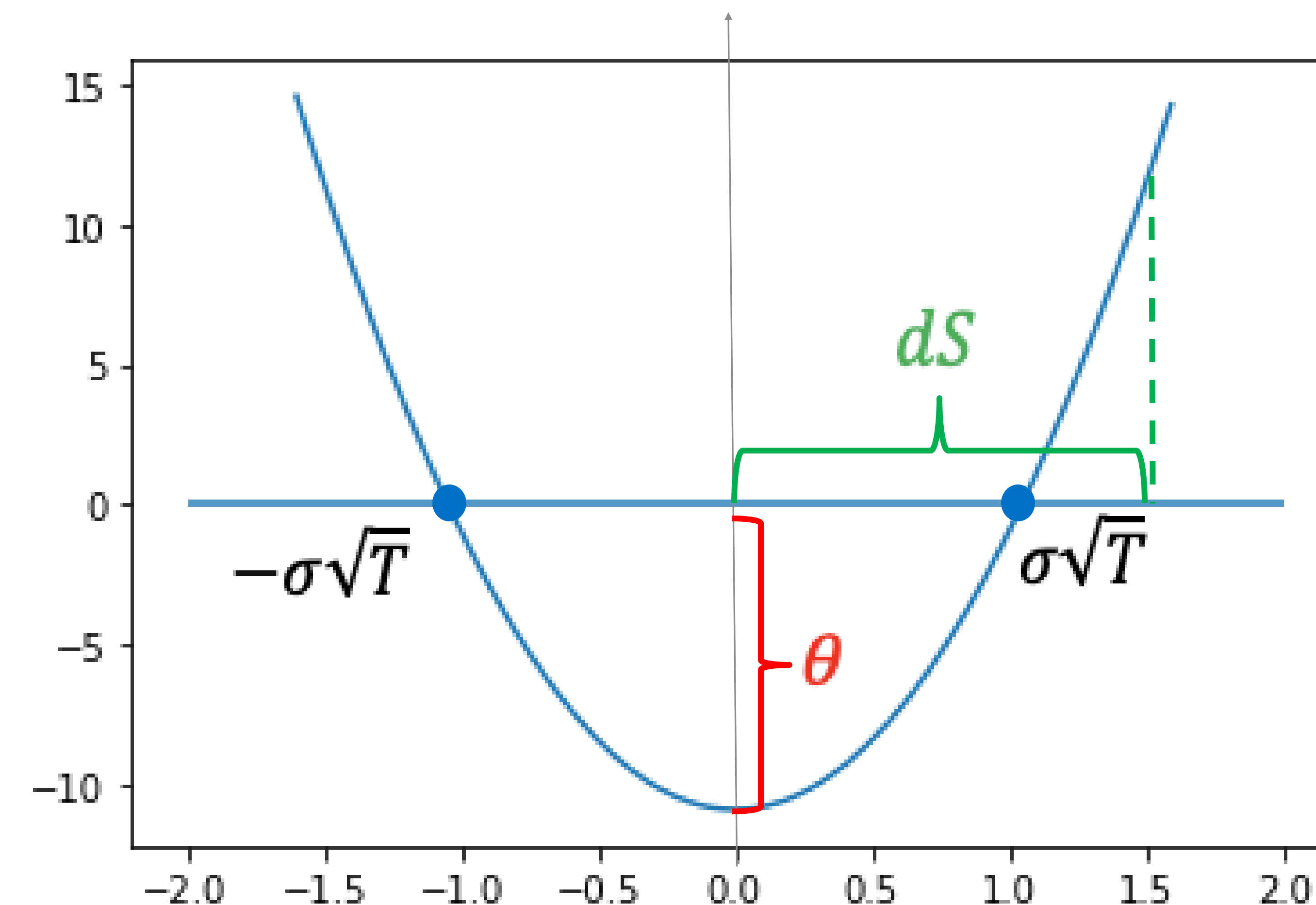
# Example #2

## Path-Dependent P&L Simulations

### Gamma-Theta P&L:

- Black-Scholes-Merton model
- Options grid: time-to-maturity & moneyness (long call)
- Delta-hedged
- Simulation horizon:  $N$  time steps
- Simulate  $M$  spot paths to horizon
- Compute gamma-theta P&L per path as the options age and their moneyness shifts

$$P\&L = \sum_{i=0}^{N-1} \left( \frac{1}{2} \Gamma_i (dS_i)^2 + \theta_i dt \right)$$



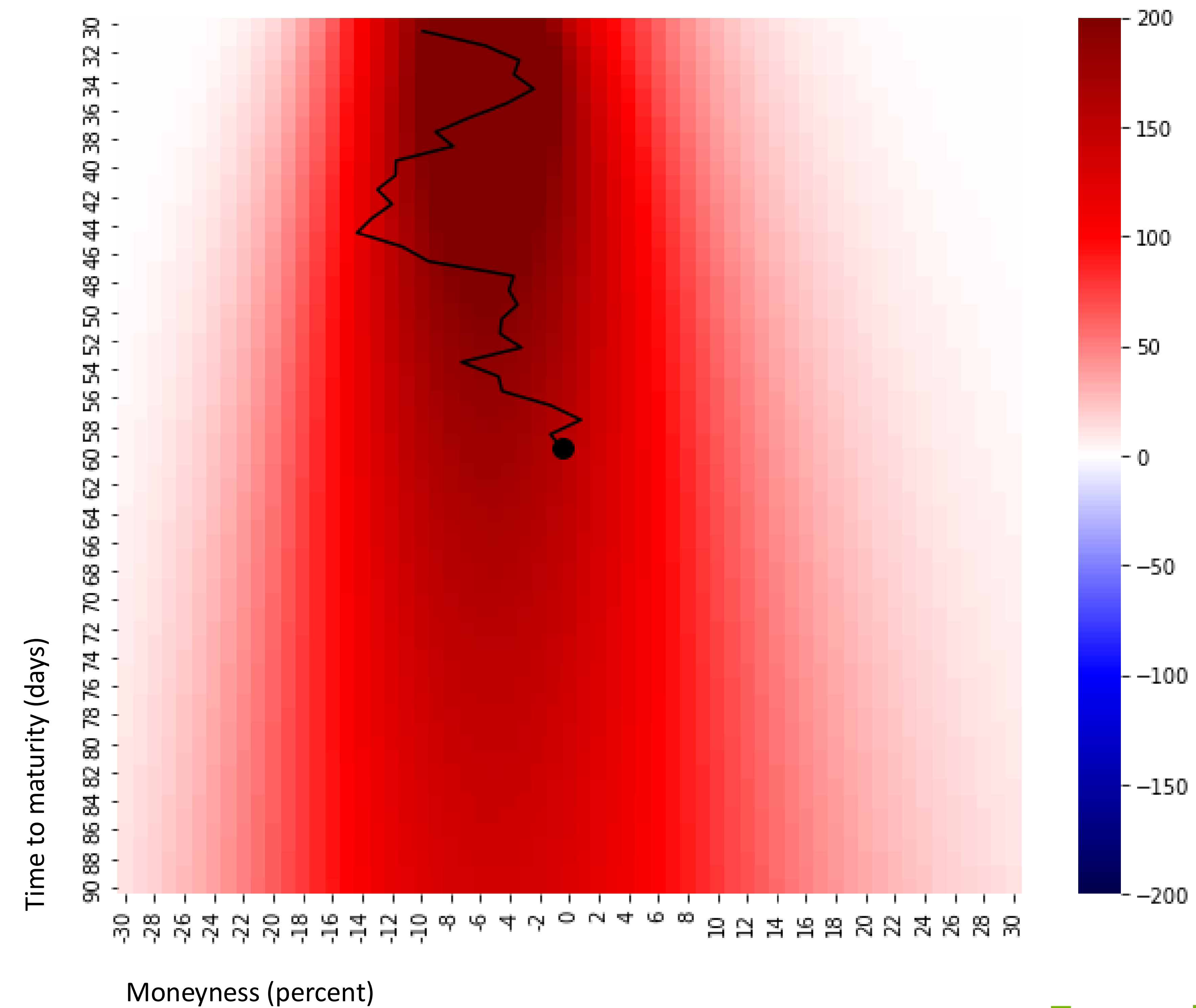
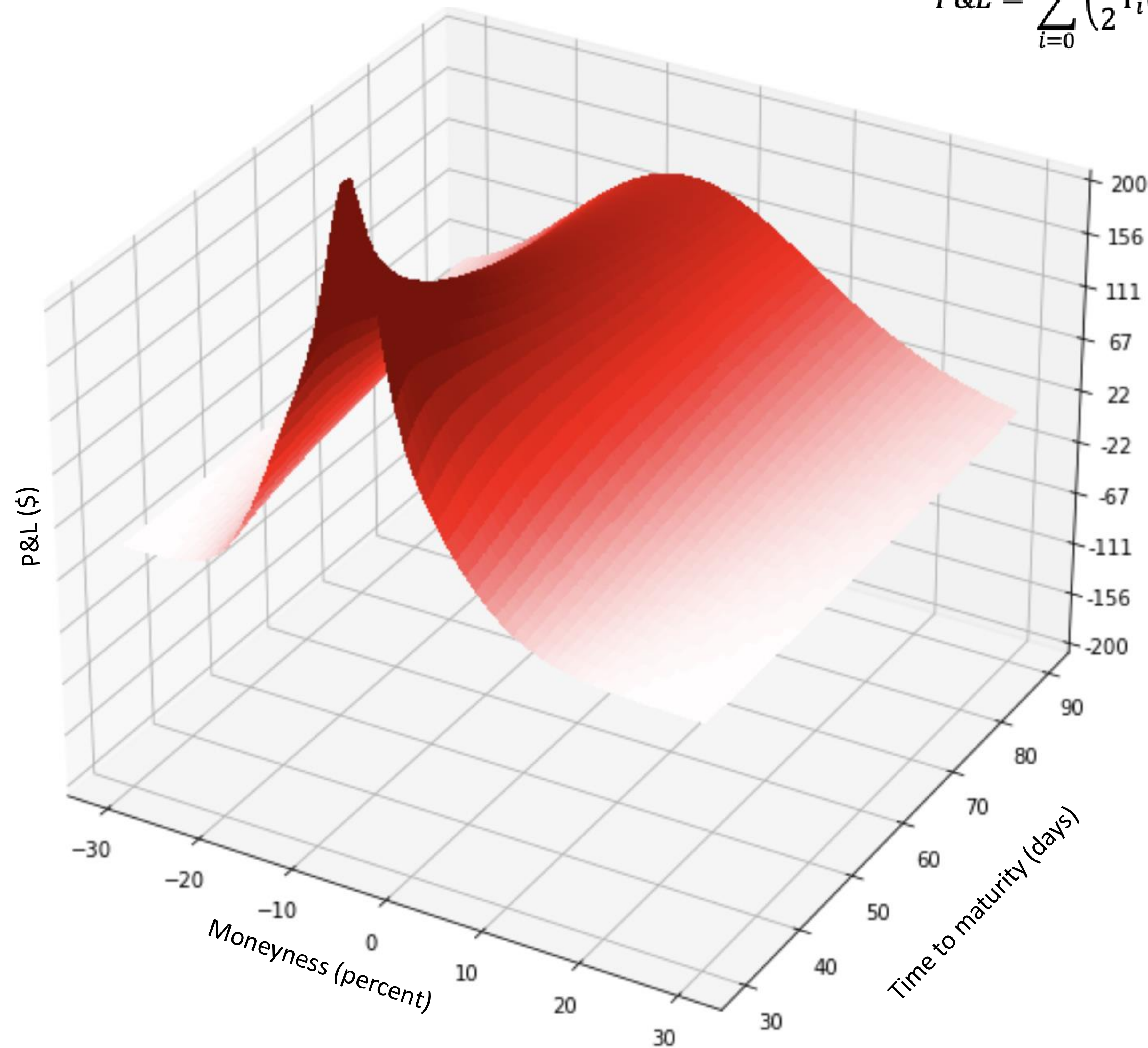


# Gamma-Theta P&L Distributions

Spot Price Simulation to Horizon

$$P\&L = \sum_{i=0}^{N-1} \left( \frac{1}{2} \Gamma_i (dS_i)^2 + \theta_i dt \right)$$

Path-dependent 30-day gamma-theta P&L

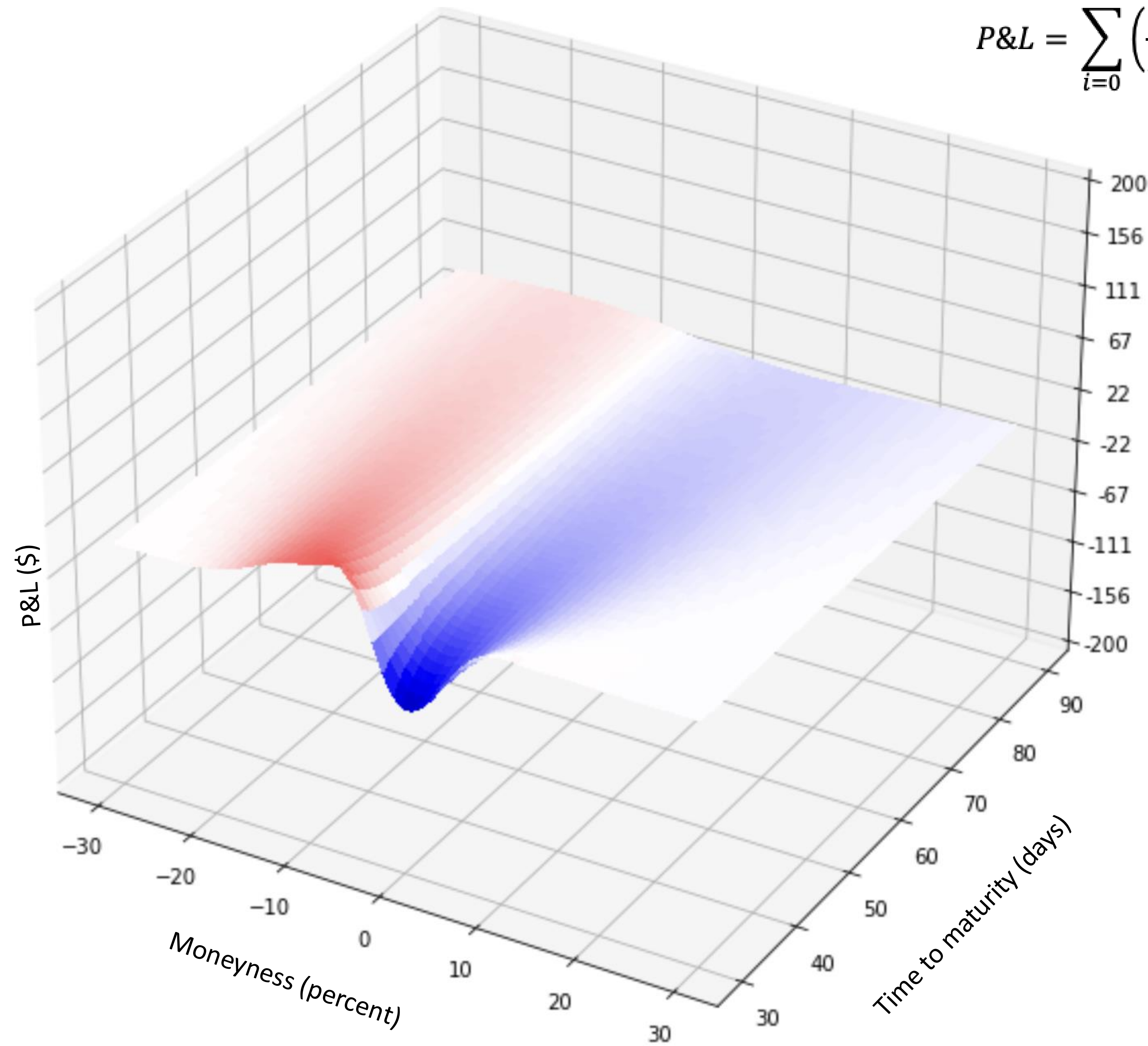




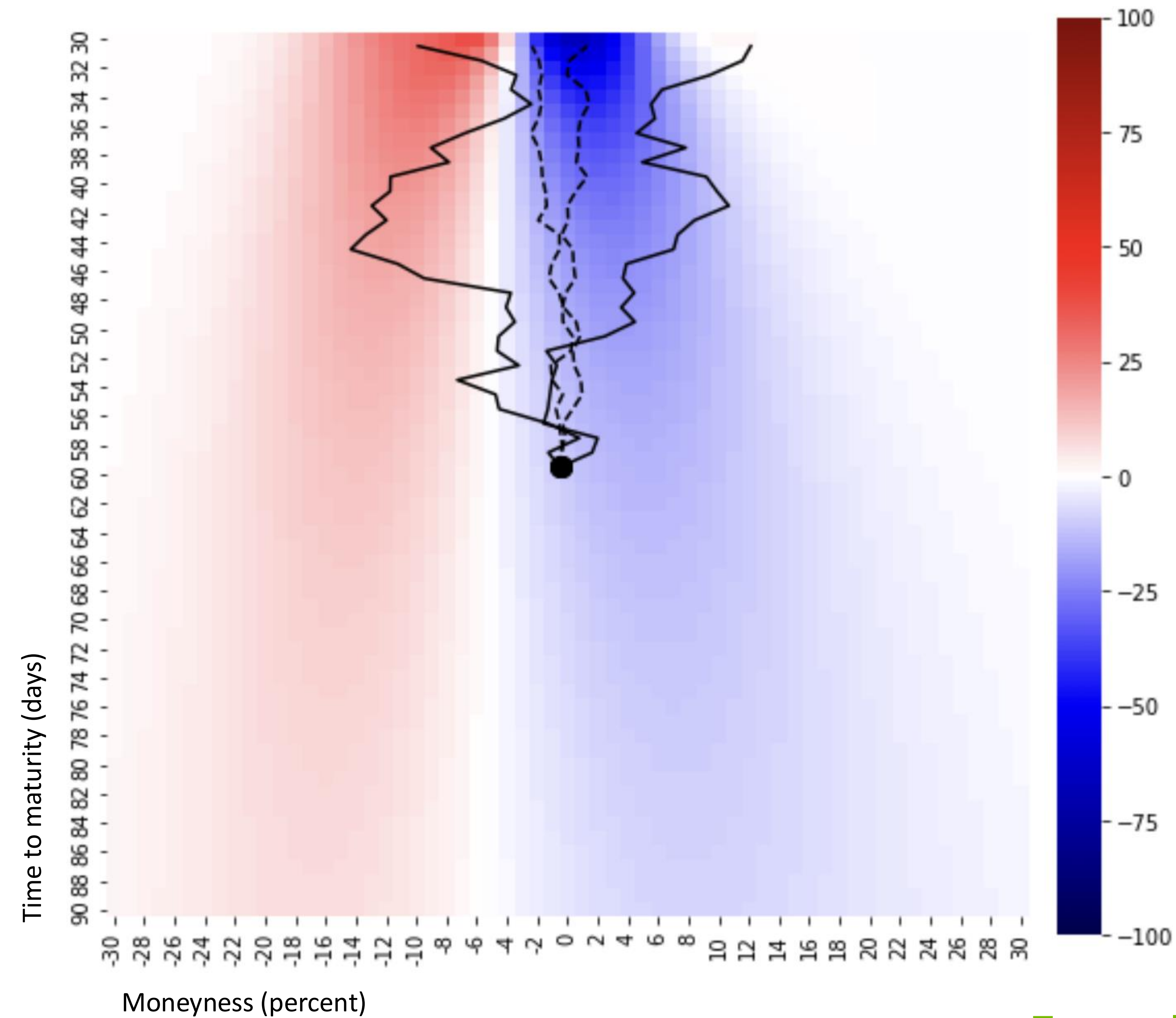
# Gamma-Theta P&L Distributions

Spot Price Simulation to Horizon

$$P\&L = \sum_{i=0}^{N-1} \left( \frac{1}{2} \Gamma_i (dS_i)^2 + \theta_i dt \right)$$



Path-dependent 30-day gamma-theta P&L





# Example 1: BSM Model in Parallel C++

## Original C code

- We started with a C code that calculates BSM using a serial for loop across options and greeks.
- The BlackScholesBody method implements pricing of Puts and Calls for one option, you've likely already written this.
- This baseline code could be CPU-parallelized using OpenMP directives.
- This code is fundamentally serial with parallelism added later via OpenMP.
- Moving this code from a multicore CPU to a GPU would require rethinking the OpenMP directives, possibly using OpenACC or even CUDA C++.

```
void BlackScholesCPUAllGreeks(
    double *h_CallResult,
    double *h_PutResult,
    double spotPrice,
    double *h_OptionStrike,
    double *h_OptionYears,
    double Riskfree,
    double *h_OptionVol,
    int optN
)
{
    #pragma omp parallel for collapse(2)
    for (int greek=0; greek < NUM_GREEKS; greek++)
    {
        for (int opt = 0; opt < optN; opt++)
        {
            int idx = greek * optN + opt;
            BlackScholesBody(
                h_CallResult[idx],
                h_PutResult[idx],
                spotPrice,
                h_OptionStrike[opt],
                h_OptionYears[opt],
                Riskfree,
                h_OptionVol[opt],
                static_cast<GREEK>(greek));
        }
    }
}
```



# Example 1: BSM Model in Parallel C++

## Parallel C++ code

- We use the same fundamental algorithm and code to calculate BSM, but start with C++ parallelism
1. C++ spans are the modern, idiomatic way to pass a view into your data.
  2. The **iota** view generates a range of values from 0 to  $\text{optN} - 1$  without storing every value in memory.
  3. The **for\_each** algorithm iterates on all values in the options range.
  4. The **par\_unseq** execution policy tells the compiler the code can be run in parallel and in any order.
- Simply modernizing from a simple, sequential loop to a parallel algorithm, reusing all other code, can yield significant benefits.

```
void BlackScholesStdParAllGreeks(
    array_view CallResult,
    array_view PutResult,
    double spotPrice,
    std::span<double> OptionStrike,           [1]
    std::span<double> OptionYears,
    double Riskfree,
    std::span<double> OptionVol,
    int optN)
{
    auto options = std::views::iota(0,          [2]
                                    optN * NUM_GREEKS);
    std::for_each(                             [3]
        std::execution::par_unseq,             [4]
        options.begin(),
        options.end(), [=](int idx)
    {
        int opt    = idx%optN;
        int greek = idx/optN;
        BlackScholesBody(CallResult(greek,opt),
                          PutResult(greek,opt),
                          spotPrice,
                          OptionStrike[opt],
                          OptionYears[opt],
                          Riskfree,
                          OptionVol[opt],
                          static_cast<GREEK>(greek));
    });
}
```



# Example 1: Before/After

## Minimal Changes Yield Significant Results

```
#pragma omp parallel for collapse(2)
for (int greek=0; greek < NUM_GREEKS; greek++)
{
    for (int opt = 0; opt < optN; opt++)
    {

        int idx = greek * optN + opt;
        BlackScholesBody(
            h_CallResult[idx],
            h_PutResult[idx],
            spotPrice,
            h_OptionStrike[opt],
            h_OptionYears[opt],
            Riskfree,
            h_OptionVol[opt],
            static_cast<GREEK>(greek));
    }
}
```

```
auto options = std::views::iota(0,
                                optN * NUM_GREEKS);
std::for_each(
    std::execution::par_unseq,
    options.begin(),
    options.end(), [=](int idx)
{
    int opt  = idx%optN; int greek = idx/optN;
    BlackScholesBody(
        CallResult(greek,opt),
        PutResult(greek,opt),
        spotPrice,
        OptionStrike[opt],
        OptionYears[opt],
        Riskfree,
        OptionVol[opt],
        static_cast<GREEK>(greek));
});
```



# Example 1: BSM Model in Parallel C++

Building with `nvc++`

- NVIDIA HPC SDK includes the `nvc++` compiler, which can be used to parallelism C++ algorithms for CPUs or GPUs.
  - NOTE: Other compilers support parallelizing these algorithms for CPUs too
- Adding either of the following options will parallelize the code:
  - `-stdpar=gpu` – Build the code for an NVIDIA GPU.
  - `-stdpar=multicore` – Build the code for a multicore CPU.
- At this time the decision of targeting a CPU or GPU is done at compile-time.

```
# Builds for GPU
$ nvc++ -fast -stdpar=gpu -c -o
BlackScholes_stdpar_gpu.o
BlackScholes_stdpar.cpp -std=c++20
```

```
# Builds for CPU
$ nvc++ -fast -stdpar=multicore -c -o
BlackScholes_stdpar_cpu.o
BlackScholes_stdpar.cpp -std=c++20
```

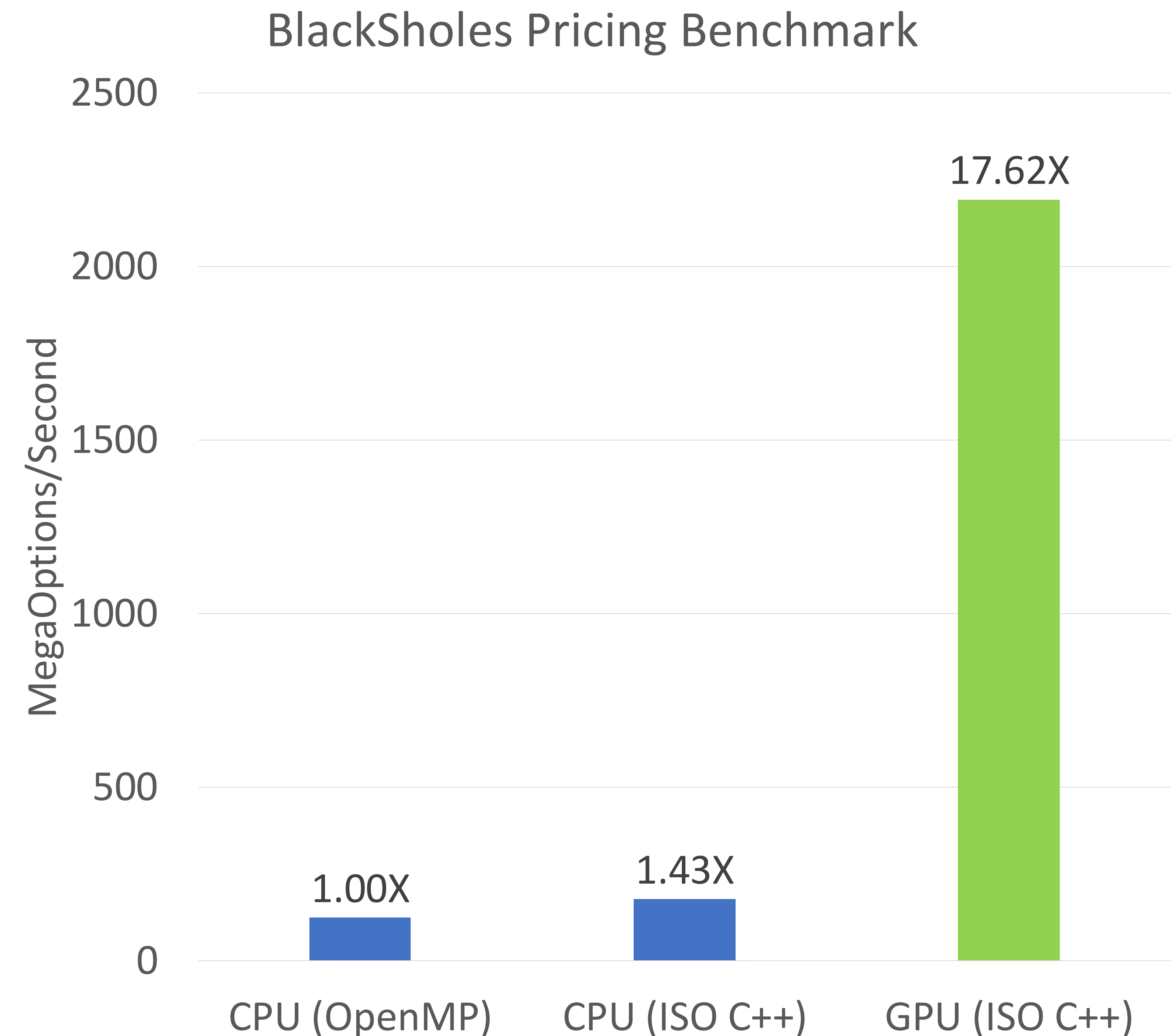
```
# Executable must be linked with the same
option as above.
$ nvc++ -o BlackScholes_gpu
BlackScholes_gold.o
BlackScholes_stdpar_gpu.o main_clean_gpu.o
-stdpar=gpu
```



# Example 1: Performance Results

## ISO C++ Delivers Portable Parallel Programming

- Using OpenMP we can already run the baseline C code on multiple CPU cores
- Moving to ISO C++ eliminated some inefficiencies to give a speed-up on the same CPU cores
- ISO C++ has the added benefit of running unmodified on the GPU for a significant speed-up.
- Nearly all code could be reused from original serial code, but refactoring to use native C++ parallelism gave significantly improved performance and portability.
- Lesson: Write parallel code!



NVIDIA HPC SDK 23.1. CPU: AMD EPYC 7742 64-Core Processor. GPU: NVIDIA A100



# Example 2: Profit & Loss Modeling

## Random Path Generation

- This example reuses the BSM code to calculate Profit & Loss along a random path.
  - The C++ standard includes a collection of random number generators that can be used.
  - Here we generate some number of random paths using a normal distribution from C++ `<random>`
    - Although on-GPU RNGs exist, we chose not to use one to keep the code completely portable.
1. Here an `array_view` is an alias to an `mdspan` with 2 dimensions.
  2. Using the C++ standard RNG maintains portable code. An RNG from `Thrust` or `cuRand` could also be used for the GPU.

```
std::vector<double>
generate_paths(double s0, const double sigma_r, const int
horizon, const double dt, const int num_paths)
{
    std::vector<double> path_vec(horizon*num_paths);
    auto path = array_view(path_vec.data(),
                             num_paths,horizon);

    double rfr = 0.0,
           div = 0.0;

    auto paths = std::views::iota(0,num_paths);
    std::for_each(paths.begin(), paths.end(),
        [&](int p)
        {
            std::random_device rd{};
            std::mt19937 generator(rd());
            generator.seed(100+p);
            std::normal_distribution<double> distribution{0.0,1.0};
            path(p,0) = s0;
            auto range = std::views::iota(1,horizon);
            std::for_each(range.begin(), range.end(),
                [&](int k){
                    double w = distribution(generator);
                    path(p,k) = path(p,k-1) * exp((rfr - div -
                        (0.5*sigma_r*sigma_r))*dt
                        + sigma_r*sqrt(dt)*w);
                });
        });
    return std::move(path_vec);
}
```

[1]

[2]



# Example 2: Profit & Loss Modeling

## Exposing Parallelism

- The baseline version for this application uses C++ parallelism, but only across options, each path is simulated sequentially.
  - Limiting the parallelism to options along the path is simple, since no two parallel threads will ever write to the same location in the PNL array.
  - This approach limits how much parallelism the GPU will execute and causes a significant number of GPU kernel launches and synchronization.
1. All parallelism is contained in this routine.

```
auto steps = std::views::iota(1, horizon);
// Iterate from 0 to num_paths - 1
auto path_itr = std::views::iota(0, num_paths);
std::for_each(path_itr.begin(), path_itr.end(),
    [=](int p)
    {
        // Iterate from 1 to horizon - 1
        std::for_each(steps.begin(), steps.end(),
            [=](int step)
            {
                calculate_pnl(pathmd(p, step),           [1]
                             pathmd(p, step-1),
                             h_OptionStrike,
                             h_OptionYears,
                             h_OptionVol, pnl,
                             dt, step);
            });
    });
// PNL holds an accumulation of PNL for all paths,
// need to divide by num_paths
std::transform(pnl.begin(), pnl.end(), pnl.begin(),
    [=](double p){ return p/num_paths; });
// Find the maximum PNL value
auto max_pnl = std::max_element(pnl.begin(),
                                pnl.end());
```



# Example 2: Profit & Loss Modeling

## Exposing MORE Parallelism in Fewer Calls

- Since each path is independent, they can be walked in parallel with some restructuring.
1. Both span and mdspan bring metadata into the routine about their dimensions
  2. Currently multi-dimensional algorithms still require some index math, but that will be fixed in the future.
  3. Since multiple paths may write to the same option in PNL, an atomic\_ref is used to avoid a race.
  4. The atomic\_ref is used to accumulate into the appropriate value of PNL for each option.

```
void calculate_pnl_paths(const_array_view paths,
                        std::span<const double>h_optionStrike,
                        std::span<const double>h_optionYears,
                        std::span<const double>h_optionVol,
                        std::span<double>pnl, double dt)
{
    int num_paths = paths.extent(0);
    int horizon   = paths.extent(1);
    int optN      = pnl.size();
    auto opts = std::views::iota(0, optN*num_paths);
    std::for_each(std::execution::par_unseq,
                  opts.begin(), opts.end(),
                  [=](int idx){
                      int path = idx/optN;
                      int opt  = idx%optN;
                      std::atomic_ref<double> elem(pnl[opt]);
                      auto path_itr = std::views::iota(1, horizon);
                      double tmpPNL = std::transform_reduce(
                          path_itr.begin(), path_itr.end(),
                          0.0, std::plus{},
                          [=](int step) {

                              // Calculate gamma & theta

                              double returnVal = 0.5 * gamma *
                                                    ds_squared + (theta*dt);

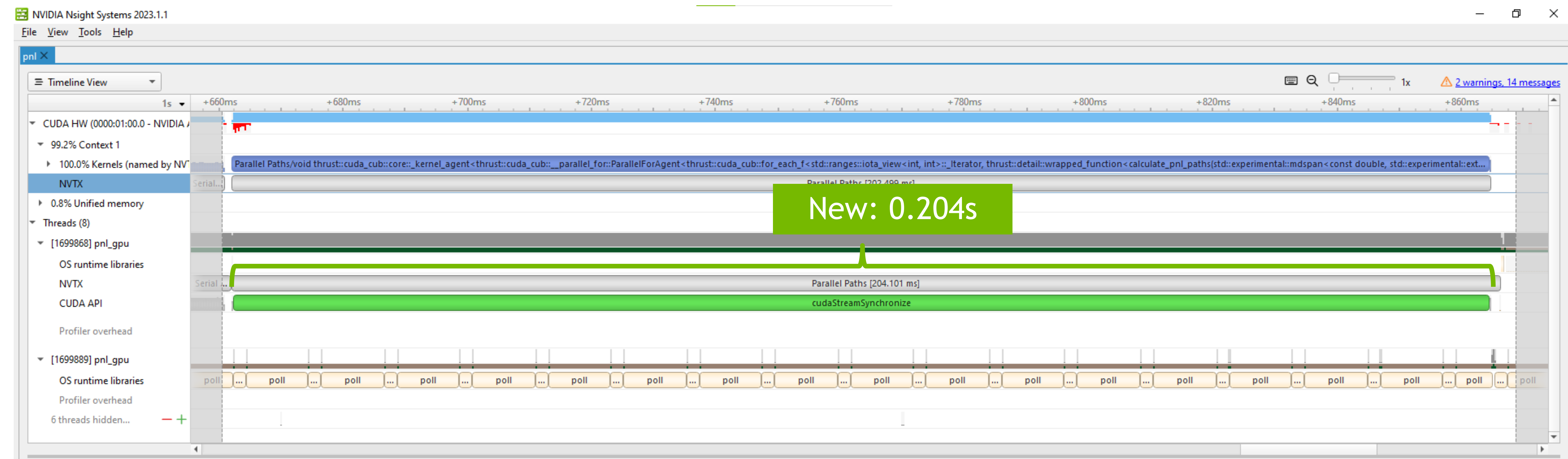
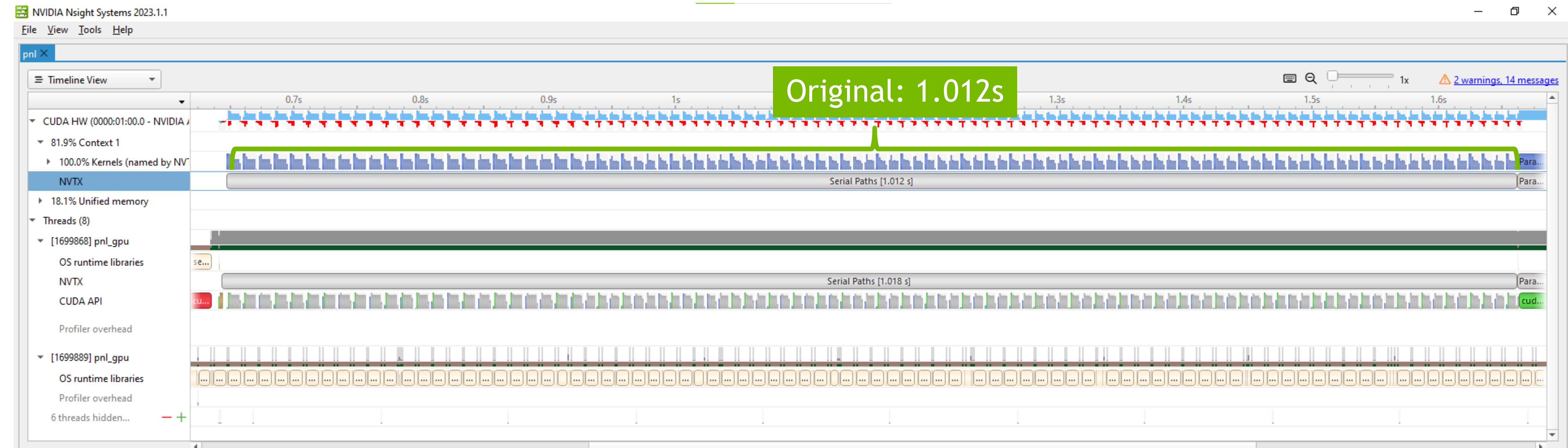
                              return returnVal;
                          });
                      elem.fetch_add(tmpPNL,
                                     std::memory_order_relaxed);
                  });
}
```



# Example 2: Profit & Loss Modeling

## Profiling The Changes

- Nsight Systems will profile standard C++ like any other GPU application
  - Original version results in hundreds of kernel launches
  - Total GPU runtime of original kernels: 1.012 seconds
- 
- Parallel paths version launches only one kernel
  - Total GPU runtime of new kernel: 0.204 seconds

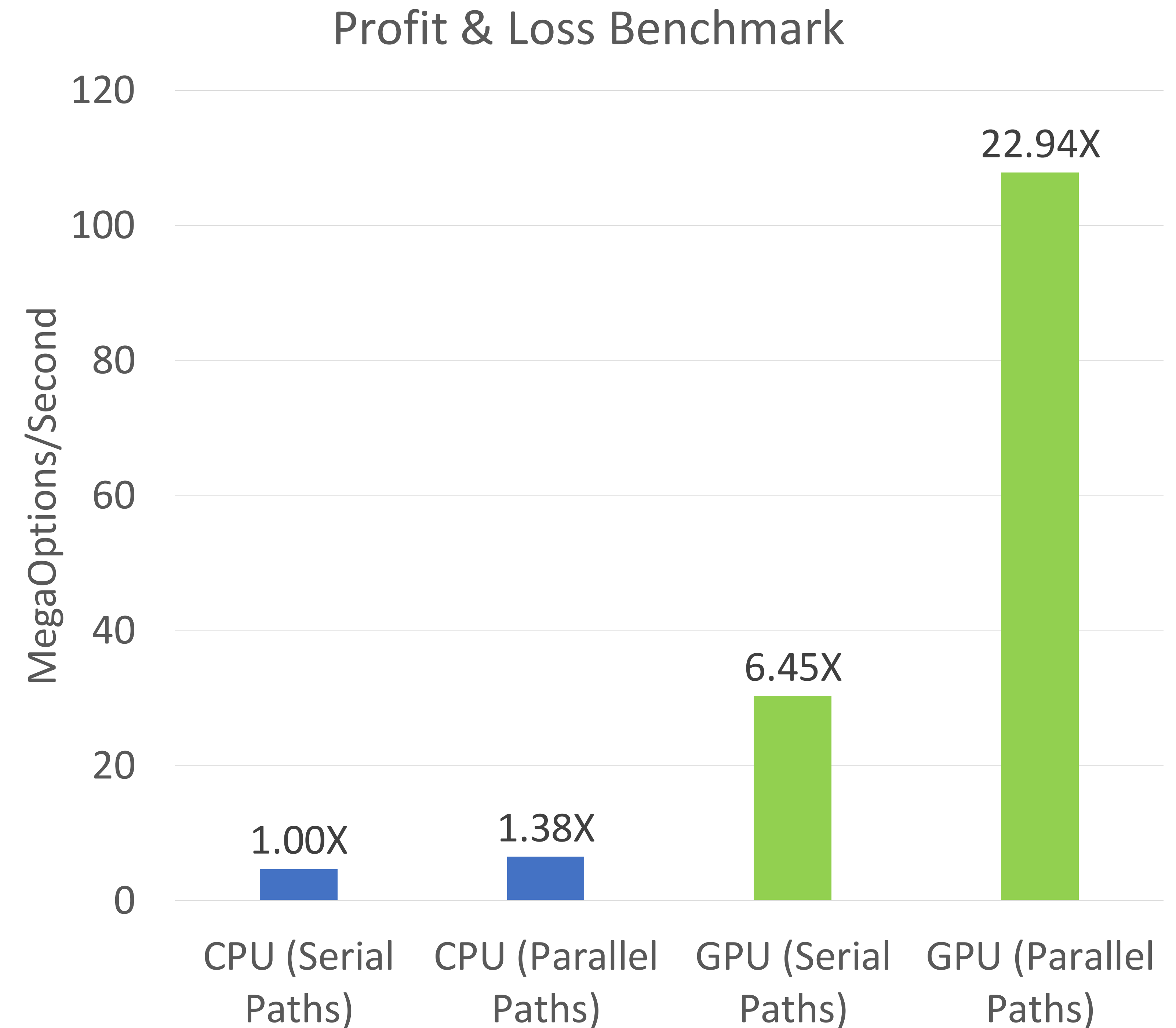




## Example 2: Performance Results

21.9M options, 180 days horizon, 100 paths

- Although the initial version used C++ parallelism, the repeated algorithms cause overheads, even on the CPU
- Even with those overheads, running on the GPU showed a benefit.
- Exposing more parallelism and reducing overheads from repeated parallel launches gave the GPU a significant advantage.
- Lesson: Expose more parallelism!



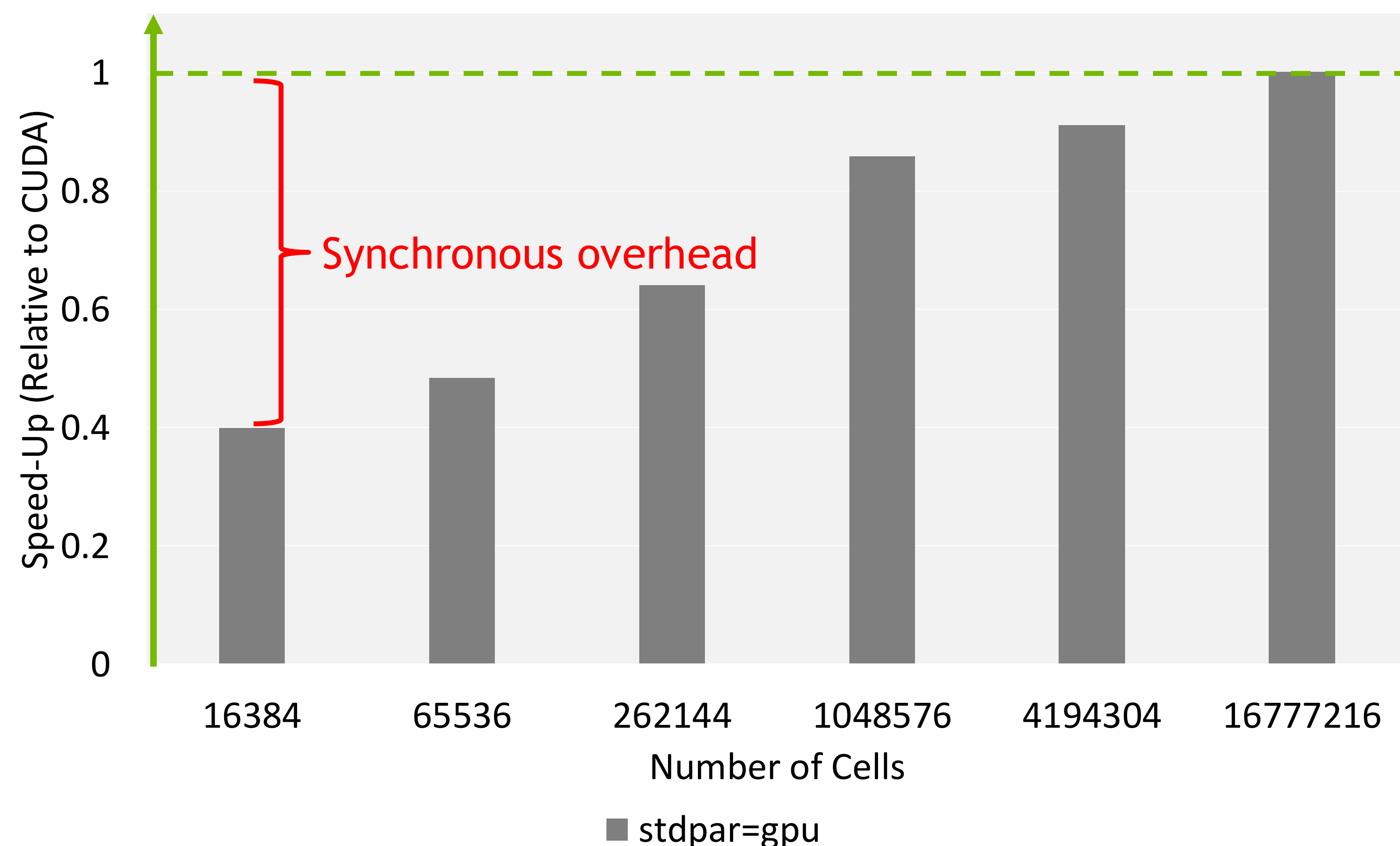
NVIDIA HPC SDK 23.1. CPU: AMD EPYC 7742 64-Core Processor. GPU: NVIDIA A100



# Standard Parallelism

## The Cost of Synchrony

- Current C++ Parallel Algorithms are synchronous, incurring extra overheads when launched on a GPU



```
// C++17 Parallel Algorithms are Synchronous
for (int step = 0; step < n_steps; step++) {
    std::for_each(par_unseq, cells_begin, cells_end, update_h);
    std::for_each(par_unseq, cells_begin, cells_end, update_e);
}

// -----

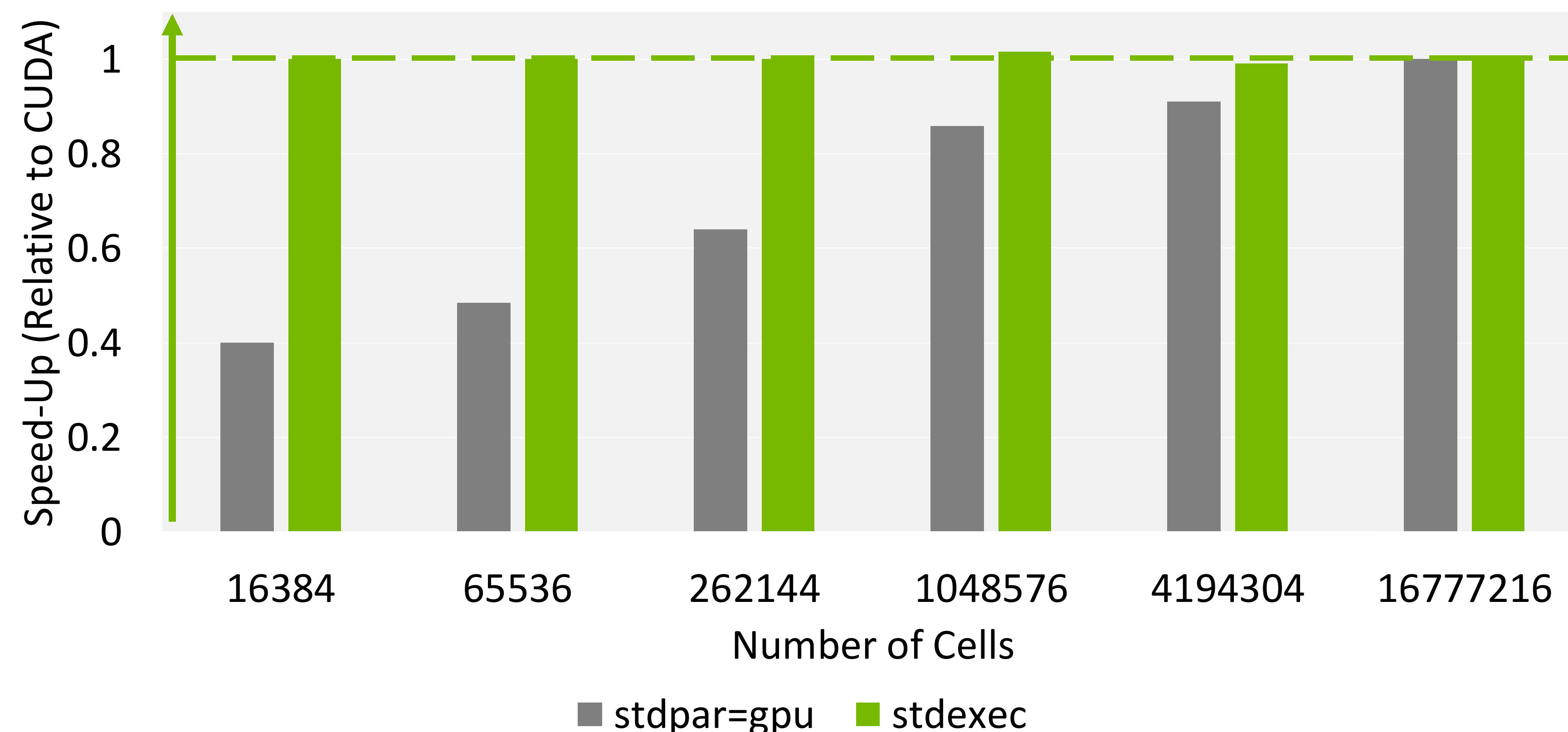
// CUDA enables asynchronously enqueueing kernels
for (int step = 0; step < n_steps; step++) {
    kernel<<<blocks, threads, 0, stream>>>(n_cells, update_h);
    kernel<<<blocks, threads, 0, stream>>>(n_cells, update_e);
}
cudaStreamSynchronize(stream);
```



# Standard Parallelism with C++ Senders

Reduce Overheads Through Asynchrony

- Senders provides a standardized means of expressing asynchronous algorithms
  - Create a sender
  - Attach a scheduler
  - Iterate within the execution graph
  - Apply bulk-parallel operations
  - Wait for the results



NVIDIA's preview of Senders/Receivers is available in HPC SDK 22.11 and newer and also [on GitHub](#).

```
// Express the algorithm as a portable execution graph
auto compute = stdexec::just() // (1)
    | exec::on( gpu_scheduler, // (2)
        nvexec::repeat_n( n_steps, // (3)
            stdexec::bulk( n_cells, update_h ) // (4)
        | stdexec::bulk( n_cells, update_e ) ) );

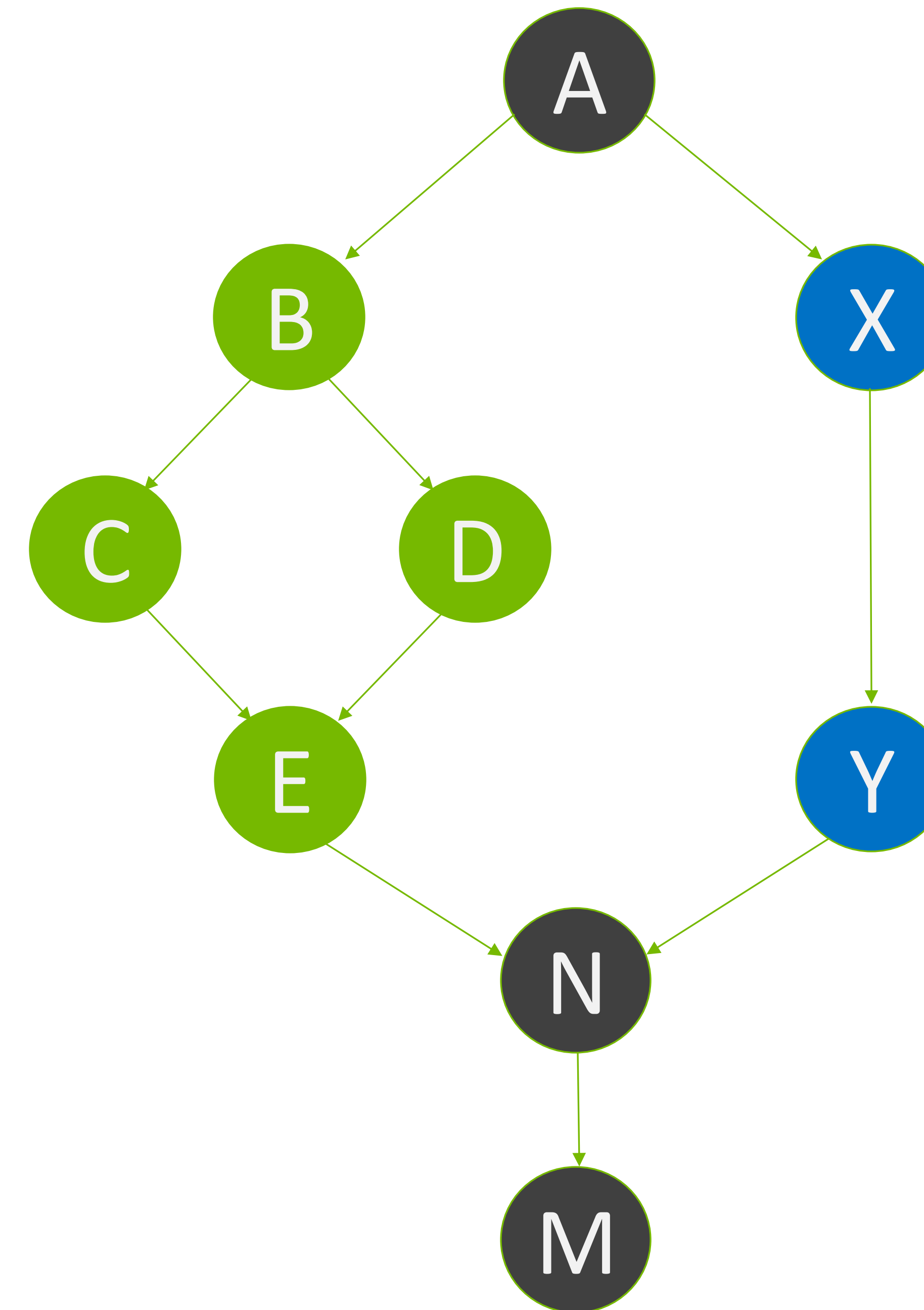
stdexec::sync_wait( std::move(compute) ); // (5)
```



# C++26 Senders/Receivers Summary

A Standard Expression of Asynchrony and Concurrency

- With Senders/Receivers it will be possible to build a task graph that can:
  - Express an algorithm without specializing it to the hardware (portable)
  - Create a task graph that can be asynchronously executed on the GPU (performant)
  - Create a hybrid task graph for CPU + GPU execution (flexible)
  - And more.
- NVIDIA has open-sourced the [stdexec library](#) to begin using Senders/Receivers today.
- See [Asynchronous Acceleration in Standard C++ \[S51755\]](#) for more information

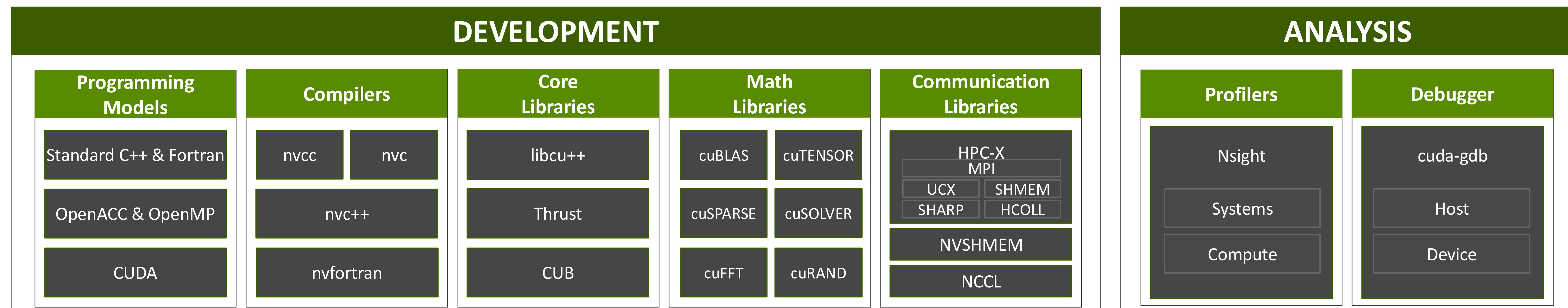




# Conclusions

## Why you should care about and begin using ISO C++ Parallelism

- ISO C++ is a mature, portable, and well-supported programming language, among the most widely used in the world.
- C++17 and beyond provide the fundamental building blocks to write parallel-first code.
- C++23 and C++26 will provide new features to further improve C++ parallel programming.
- NVIDIA HPC SDK is freely available and enables write code that's productive, portable, and performant.
  - CPU and GPU compilers, libraries, and tools
  - x86, Arm, OpenPower, all major clouds
- Writing code that's parallel first
  - Simplifies code maintenance,
  - Enables it to run on virtually any platform,
  - And improves programmer productivity.







# "Speed of Light" Valuation of Option Portfolios with cuNumeric

Manolis Papadakis, Senior Software Engineer | Nvidia GTC Live Special Event/March 21, 2023



# cuNumeric

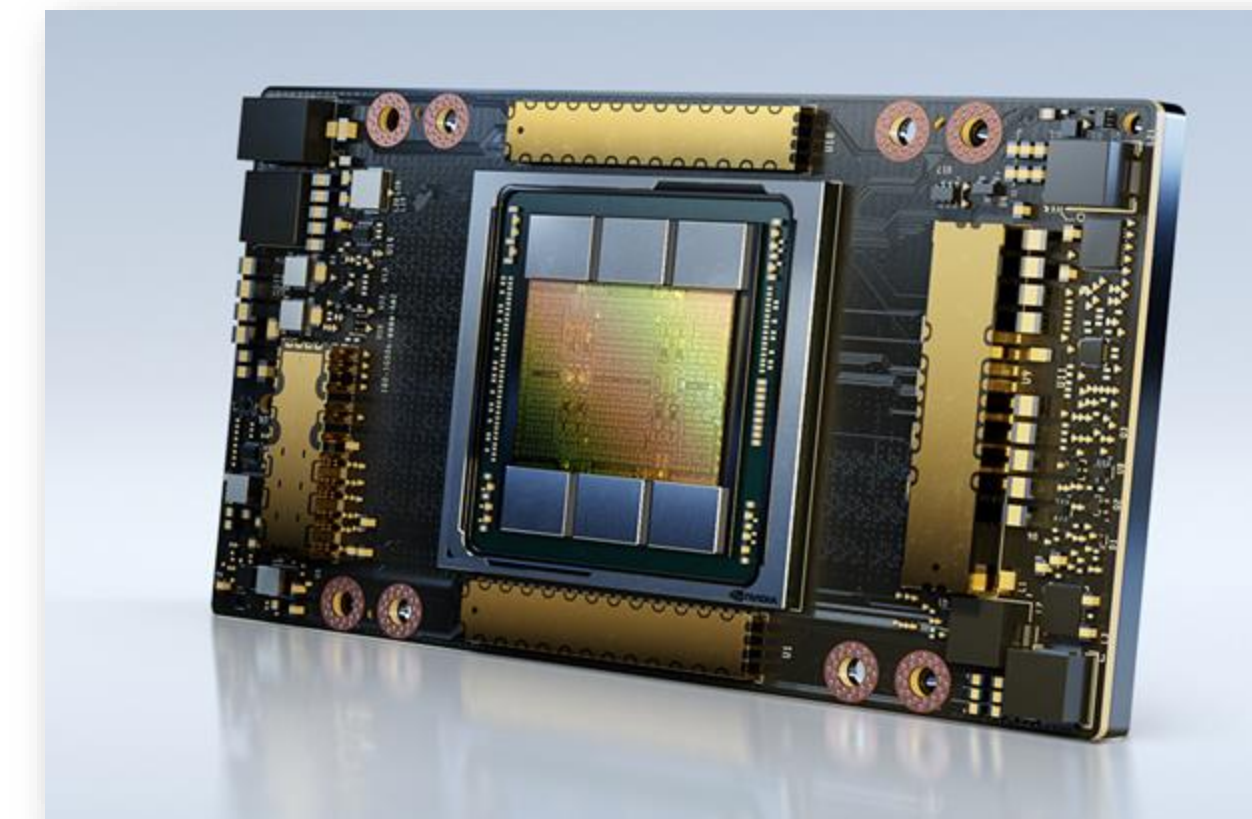
Distributed and GPU-accelerated NumPy-like

- Prototype in Python, scale up transparently
- cuNumeric automatically extracts parallelism from NumPy operators
- Get most of the benefit of parallelization without rewriting from scratch

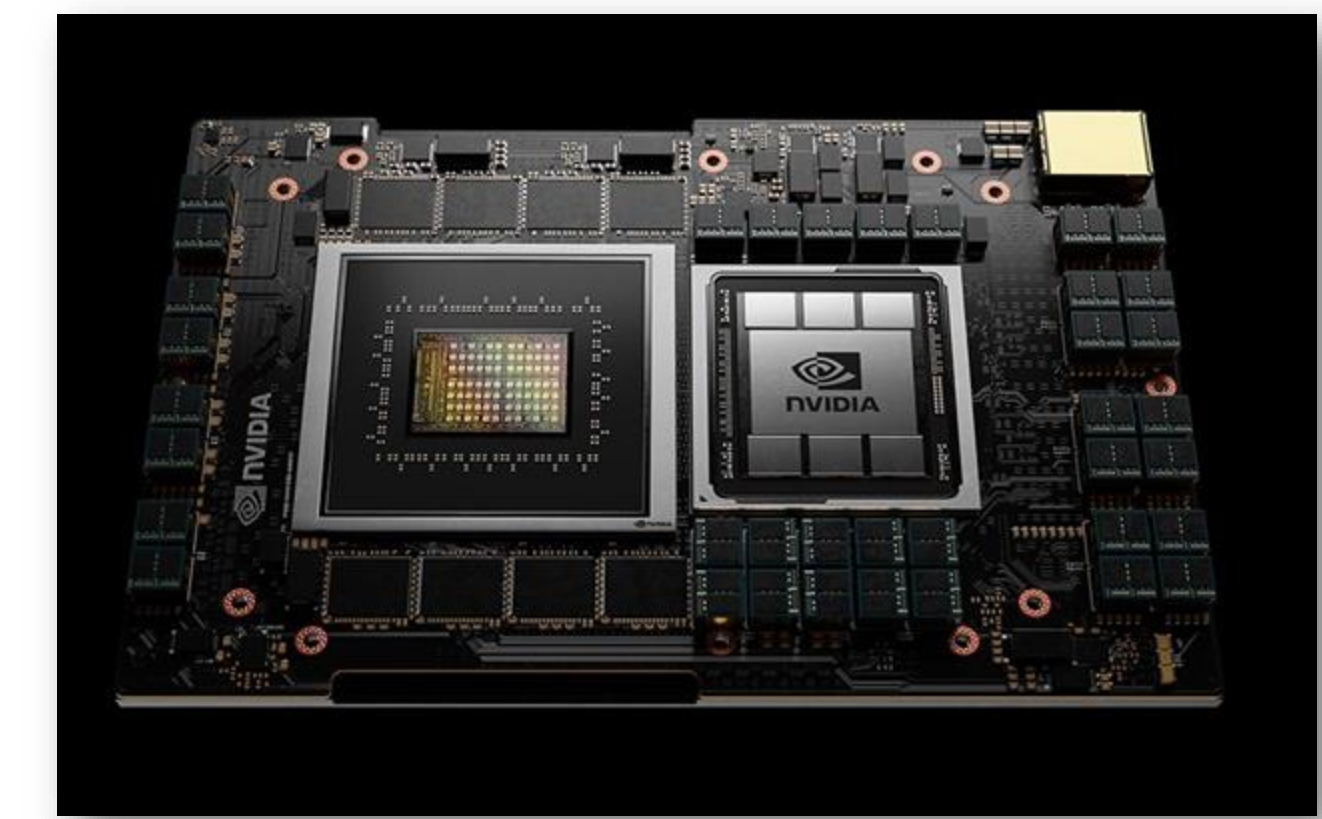
Write this

```
def cg_solve(A, b, tolerance):  
    x = np.zeros_like(b)  
    r = b - A.dot(x)  
    p = r  
    rsold = r.dot(r)  
    max_iters = b.shape[0]  
    for i in range(max_iters):  
        Ap = A.dot(p)  
        alpha = rsold / (p.dot(Ap))  
        x = x + alpha * p  
        r = r - alpha * Ap  
        rsnew = r.dot(r)  
        if np.sqrt(rsnew) < tolerance:  
            break  
        beta = rsnew / rsold  
        p = r + beta * p  
        rsold = rsnew
```

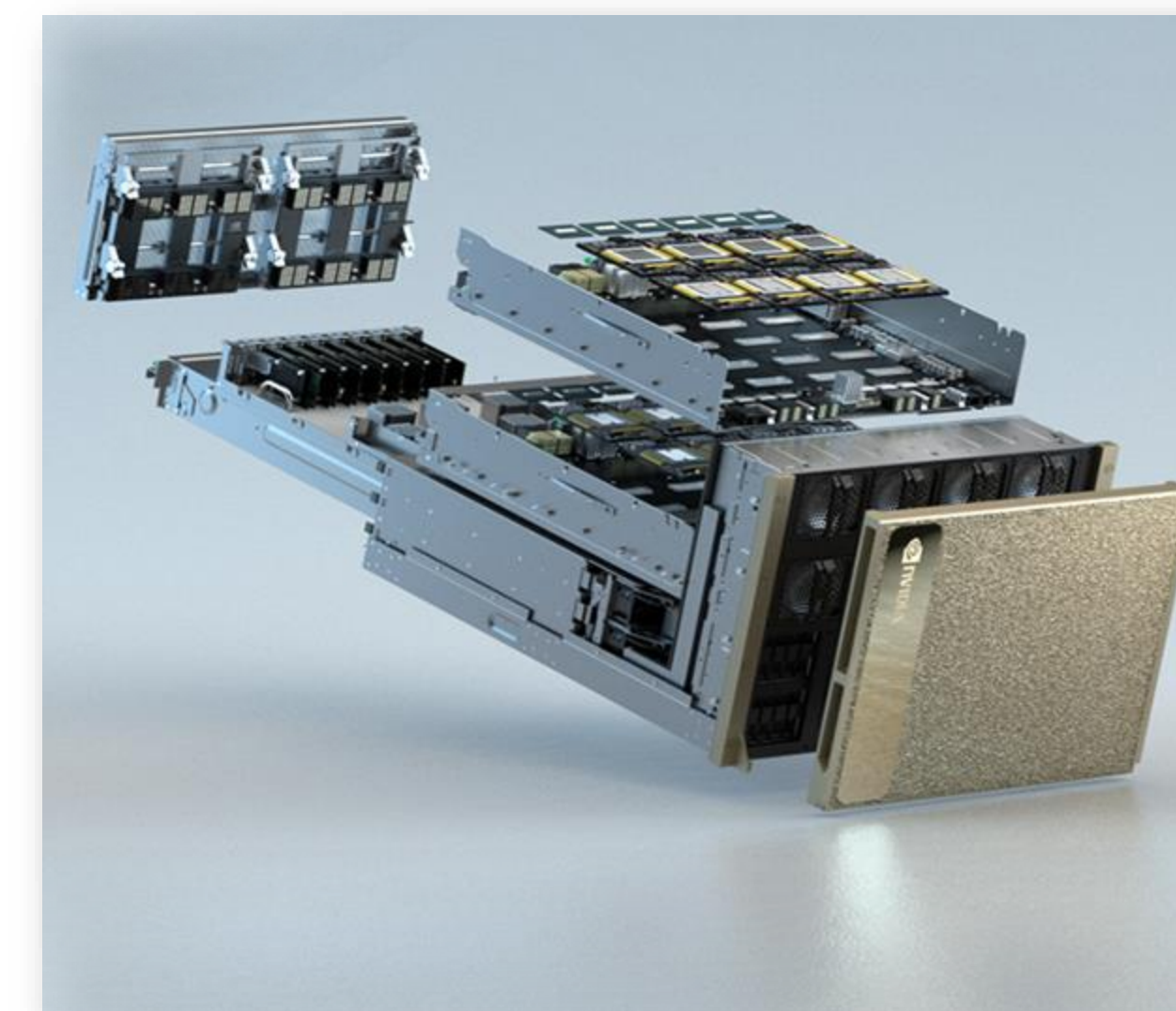
Run here



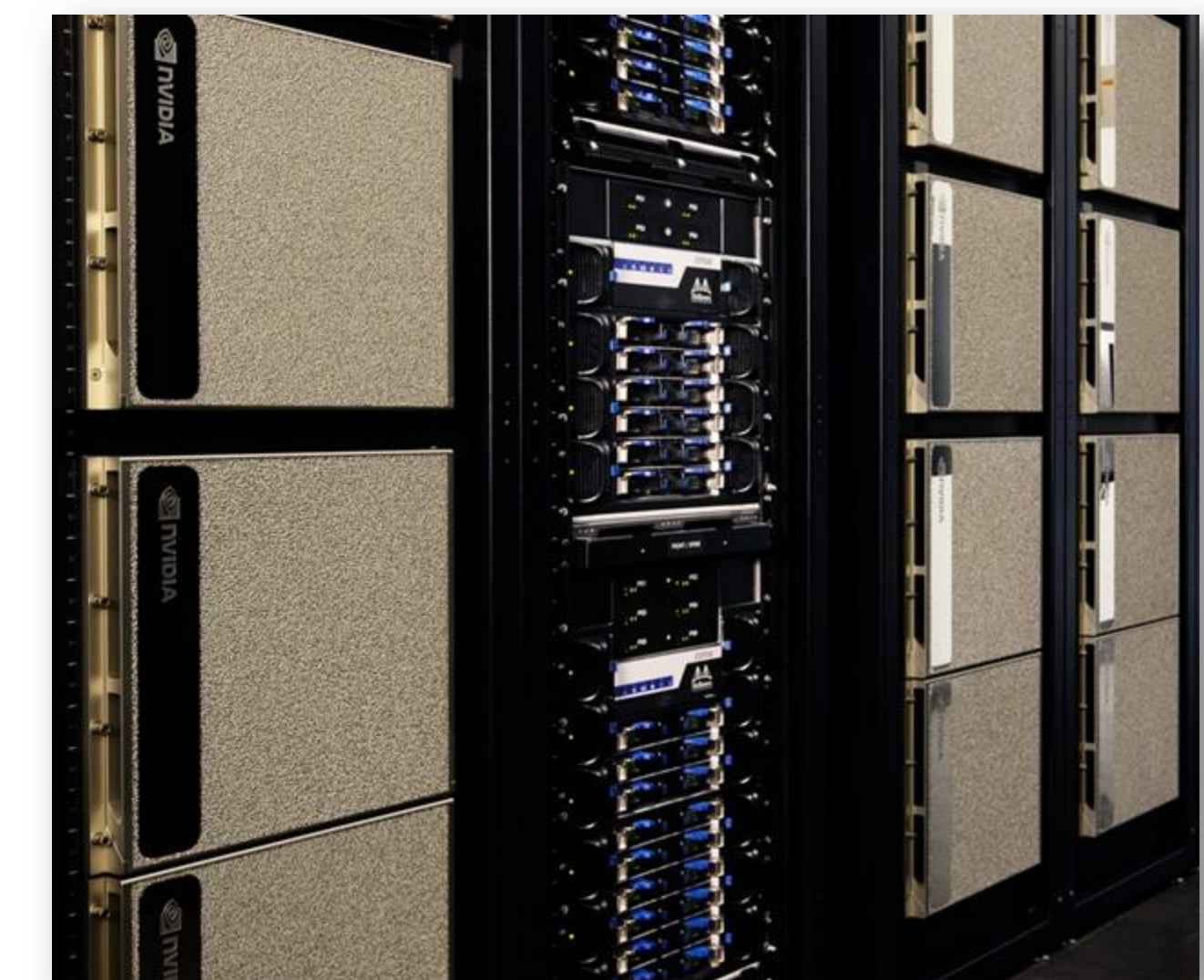
GPU



Grace CPU



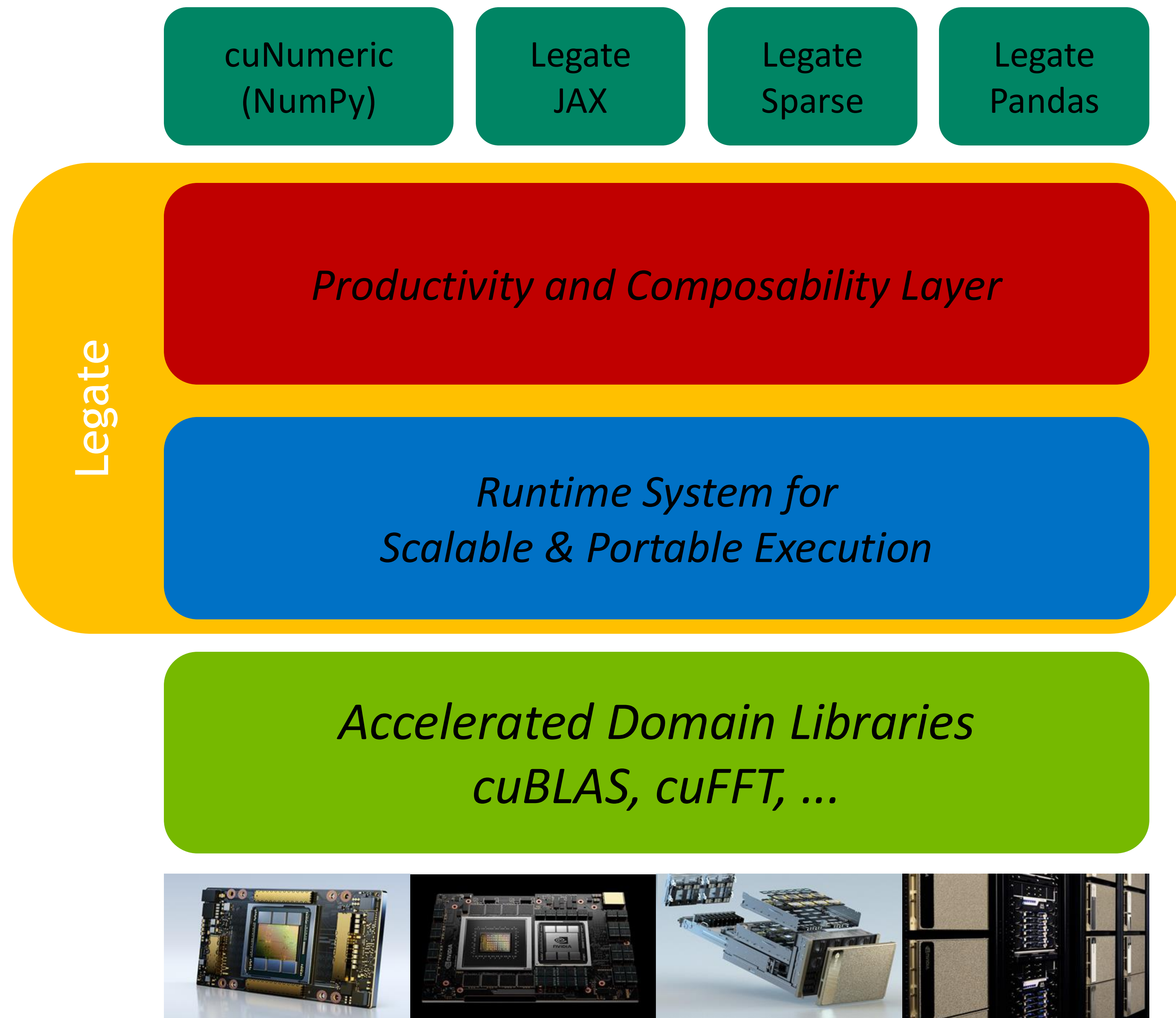
DGX



DGX SuperPOD



# cuNumeric in the Legate ecosystem



*Ecosystem of implicitly parallel libraries* that are composable and easy-to-use

*Productivity layer* that accelerates library development

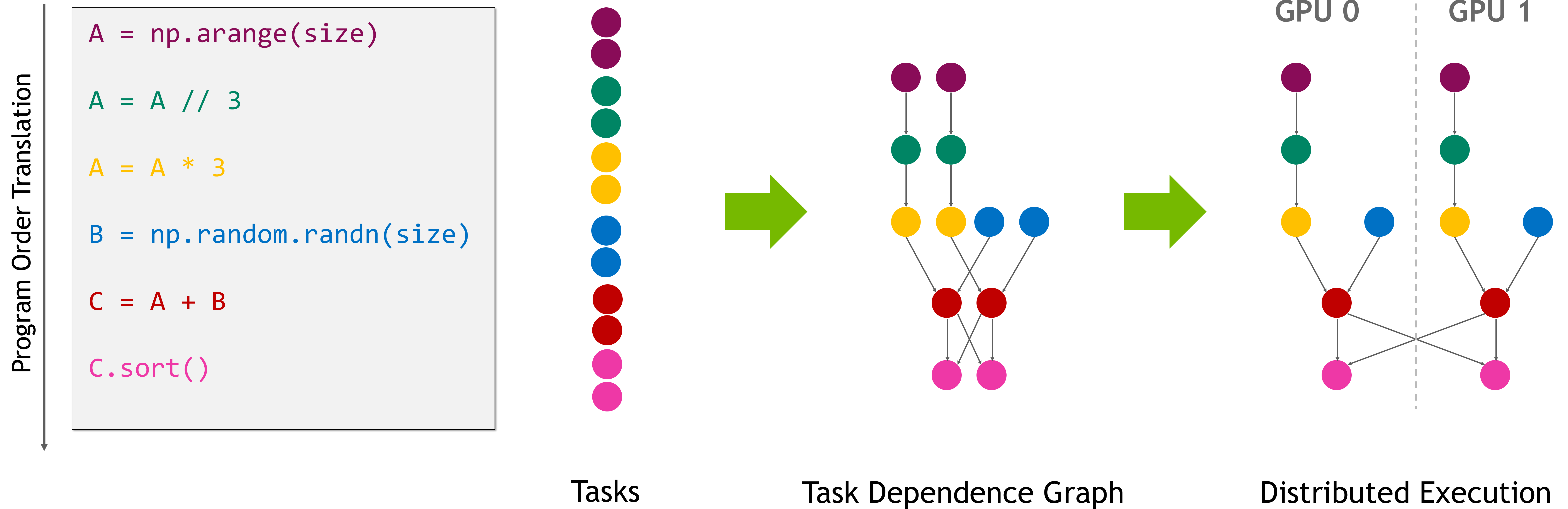
*Common runtime system* for scalable extraction of implicit parallelism

*Accelerated domain libraries* for excellent single-accelerator performance

*Nvidia hardware*



# cuNumeric program in action





# Using cuNumeric

- Install cuNumeric through conda:  

```
$ conda install -c nvidia -c conda-forge -c legate cunumeric
```
- Replace import statement:  

```
import cunumeric as np
```
- Run a script:  

```
$ legate --gpus 2 --fbmem 15000 my_program.py
```

or

```
$ LEGATE_CONFIG="--gpus 2 --fbmem 15000" python my_program.py
```
- Run in a notebook:  

```
$ legate-jupyter --gpus 2 --fbmem 15000
```

```
$ jupyter notebook --port=8888 --no-browser
```
- Run multi-node (currently requires source build):  

```
$ legate --nodes 2 --launcher mpirun --gpus 2 --fbmem 15000 my_program.py
```
- Profile execution:  

```
$ legate --gpus 2 --fbmem 15000 --profile my_program.py
```



# Code comparison (C++/Python)

## Black/Scholes model

```
inline double black_scholes(
    double s, double k, double r, double t,
    double v, int cp, const GREEK &greek) {

    const double EPS = 1e-8;
    double inf = std::numeric_limits<double>::infinity();
    double stdev = v * sqrt(t);
    double d1 = 0., d2 = 0., nd1 = 0., nd2 = 0.;
    double df = exp(-r * t);
    d1 = (log(s / k) + (r + 0.5 * v * v) * t) / stdev;
    d2 = d1 - stdev;
    nd1 = normCDF(cp * d1);
    nd2 = normCDF(cp * d2);

    switch (greek) {
    case PREM:
        return cp * (s * nd1 - k * df * nd2);
    case DELTA:
        return cp * nd1;
    case VEGA:
        return s * sqrt(t) * normPDF(d1);
    case GAMMA:
        return normPDF(d1) / (s * v * sqrt(t));
    case VANNA:
        return -d2 * normPDF(d1) / v;
    case VOLGA:
        return s * sqrt(t) * d1 * d2 * normPDF(d1) / v;
    case THETA:
        return -(0.5 * s * v / sqrt(t)
            * normPDF(d1) + cp * r * df * k * nd2);
    default:
        return 0.;
    }
}
```

```
def black_scholes(out, S, K, R, T, V, CP, greek):
```

```
    EPS = 0.00000001
```

```
    stdev = V * np.sqrt(T)
```

```
    df = np.exp(-R*T)
```

```
    d1 = (np.log(S/K)+(R+0.5*V*V)*T)/stdev
```

```
    d2 = d1 - stdev
```

```
    nd1 = normCDF(CP*d1)
```

```
    nd2 = normCDF(CP*d2)
```

```
    if greek == Greeks.PREM:
```

```
        out[...] = CP*(S*nd1 - K*df*nd2)
```

```
    elif greek == Greeks.DELTA:
```

```
        out[...] = CP*nd1
```

```
    elif greek == Greeks.VEGA:
```

```
        out[...] = S*np.sqrt(T)*normPDF(d1)
```

```
    elif greek == Greeks.GAMMA:
```

```
        out[...] = normPDF(d1)/(S*V*np.sqrt(T))
```

```
    elif greek == Greeks.VANNA:
```

```
        out[...] = -d2*normPDF(d1)/V
```

```
    elif greek == Greeks.VOLGA:
```

```
        out[...] = S*np.sqrt(T)*d1*d2*normPDF(d1)/V
```

```
    elif greek == Greeks.THETA:
```

```
        out[...] = -(0.5*S*V/np.sqrt(T)*normPDF(d1)+CP*R*df*K*nd2)
```

```
    else:
```

```
        raise RuntimeError("Wrong greek name is passed")
```

Scalar operations  
will also work  
array-wide



## Vectorized operations parallelized automatically

```
CALL = np.zeros((N_GREEKS, n_t_steps, n_vol_steps, n_money_steps), dtype = D)  
PUT = np.zeros((N_GREEKS, n_t_steps, n_vol_steps, n_money_steps), dtype = D)
```

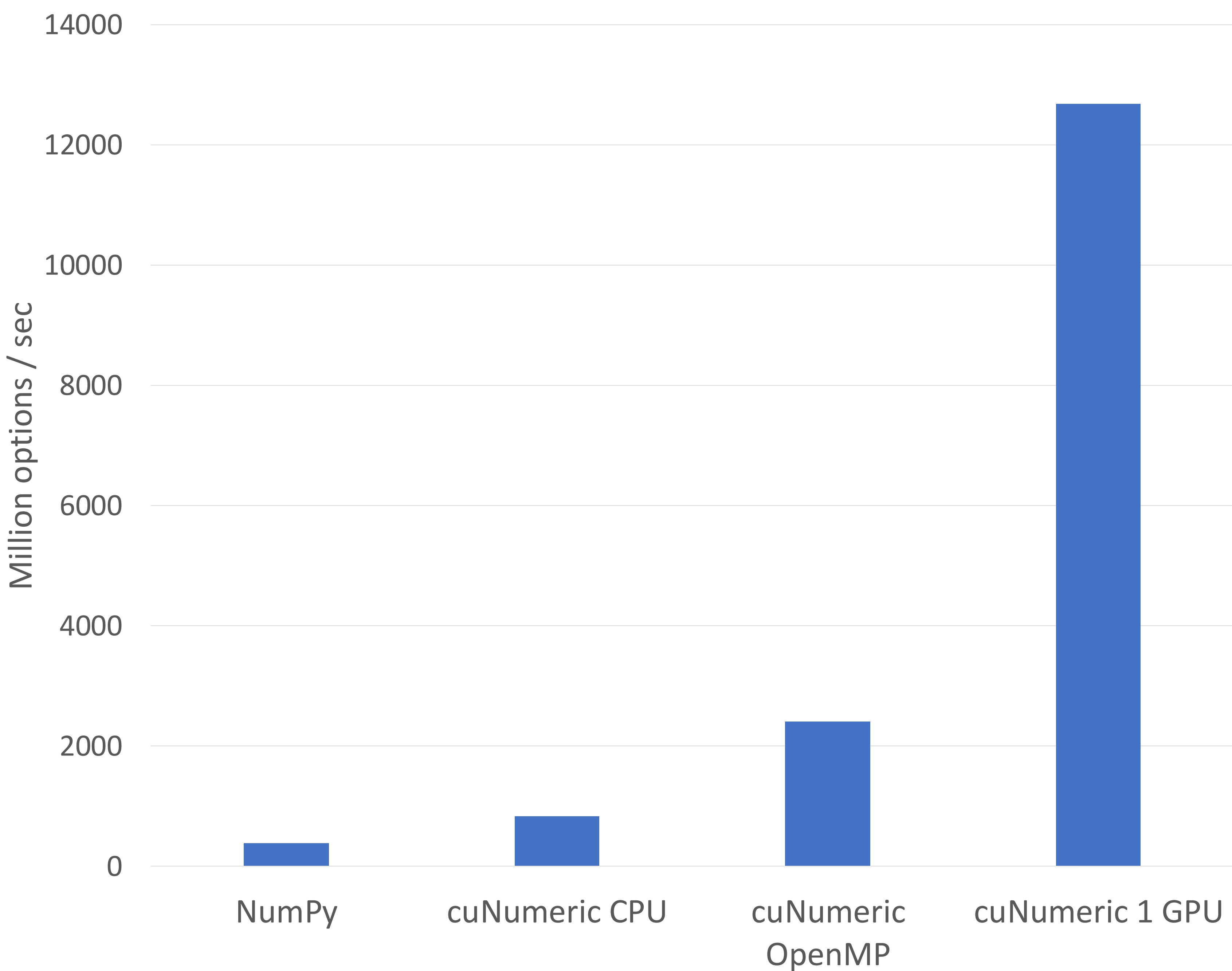
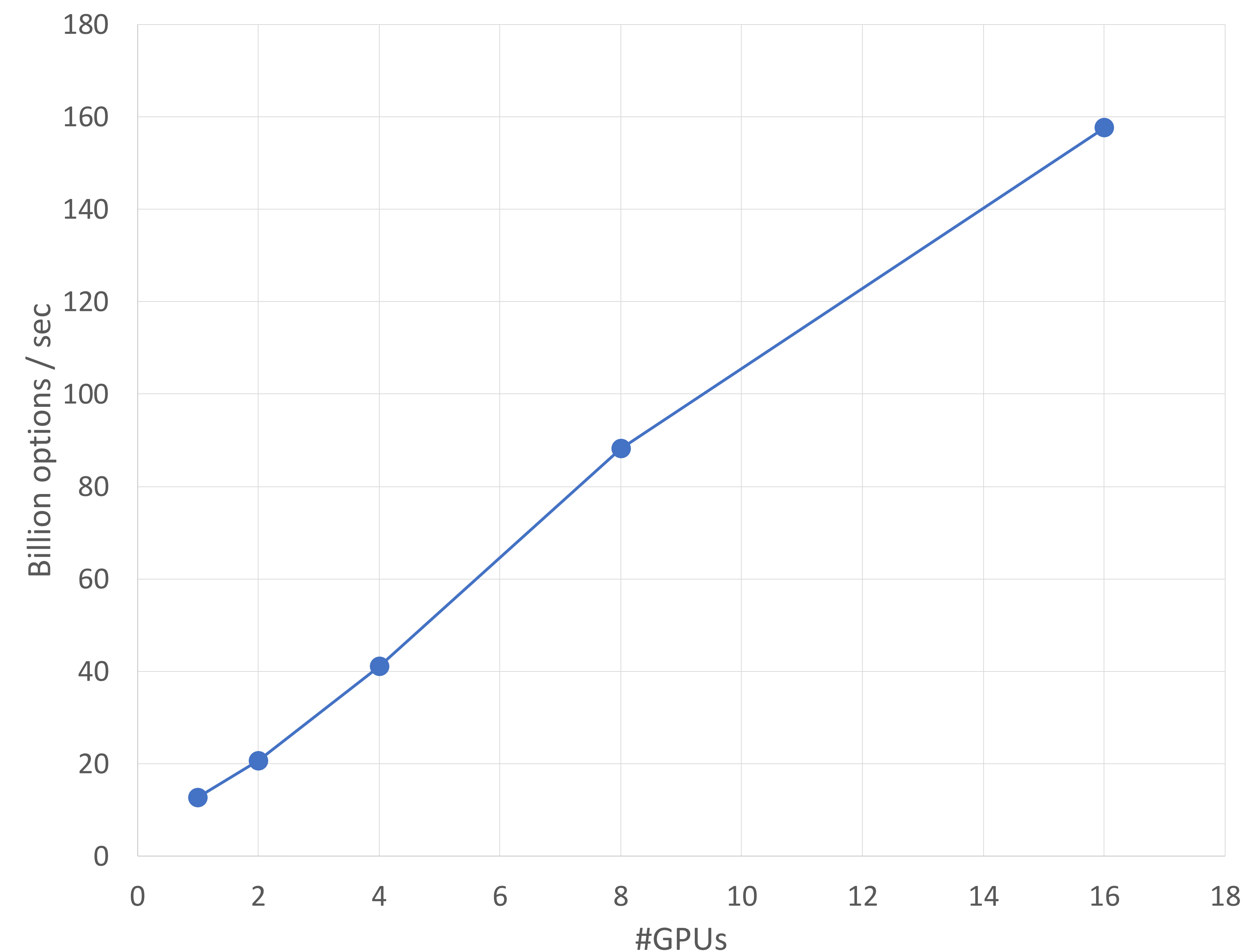
```
...
```

```
for g in Greeks:  
    # Each CALL[g] and PUT[g] is an array of calls/puts  
    # Each "black_scholes" call is parallelized across options  
    black_scholes(CALL[g.value], S, K, R, T, V, 1, g)  
    black_scholes(PUT[g.value], S, K, R, T, V, -1, g)
```



# cuNumeric performance

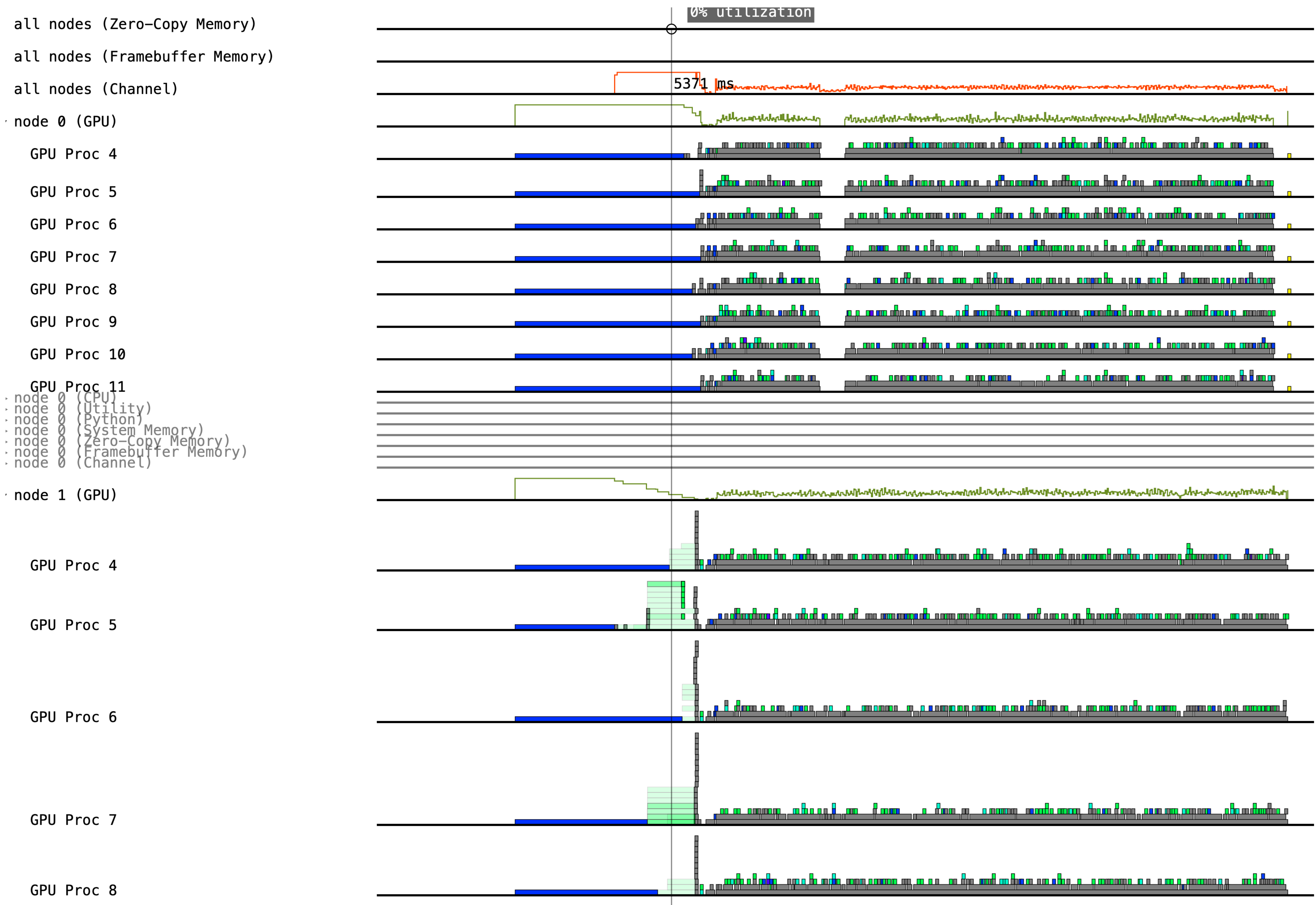
Black-Scholes pricing benchmark



experiments done on DGX-1V



# Profile (2-node run, 8 GPUs/node)





# Getting good performance out of cuNumeric

- No unaccelerated function inside inner loop  
e.g. had to provide implementation for normCDF
- cuNumeric can't parallelize for-loops / comprehensions; use array operations

<pre>sum = 0 for x in arr:     sum += x</pre>	→	<pre>sum = np.sum(arr)</pre>
---	---	------------------------------

- Conditionals in inner loop won't work with arrays
  - Write code in vectorized style:  
`out = np.where(in < 0, in + 1, in + 2)`
  - Upcoming feature:  
`outputs = np.vectorize(bar)(inputs)`

```
def bar(x):
    if x < 0:
        return x + 1
    else:
        return x + 2

in = np.array(...)
out = bar(in) # bad
```



# cuNumeric: Current status

## 60% API coverage

- Advanced indexing
- Tensor contraction
- Multi-dimensional sorting
- 96% of ufuncs
- 80% of RNGs

## Improved ergonomics

- NumPy interop/fallback
  - Seamless interoperability via array interface

```
a = numpy.zeros(10)
b = cunumeric.ones(10)
numpy.add(a, b) # dispatches to cuNumeric
```

- Partially implemented operations fall back to NumPy
- API coverage tool
- Zero code-change patching
- Conda packages
- Jupyter notebook support



# Call to Action

- Give cuNumeric a try! <https://github.com/nv-legate/cunumeric>
- Tune in for the Legate/cuNumeric GTC talk:  
[S51789]  
**“cuNumeric and Legate: How to Create a Distributed GPU Accelerated Library”**  
Thursday, Mar 23, 8:00 AM PDT
- Contact us at [legate@nvidia.com](mailto:legate@nvidia.com) with your use cases





# LLMs for Language and Beyond

Emanuel Scoullos, PhD – Sr. Solutions Architect | Nvidia GTC Live Special Event/March 21, 2023



# A LLM is worth a thousand words

LLMs in the Financial Services Industry open new opportunities and accelerate time to insight

- LLMs are transforming FSI with **more accurate** and sophisticated **predictive** models
- LLMs already used for translation, text summarization and sentiment analysis
- More use cases continue to be found; models keep getting more accurate.
- In Finance, LLMs are being used to analyze financial data including:
  - News articles
  - Earnings Calls & Reports
  - Social Media posts
  - Customer Feedback
  - Equity data
  - Macroeconomic trends
  - Credit Card transactions

to generate **insights**, uncover **trends**, and produce **forecasts** that can inform investment decisions





# LLMs enhance productivity

OpenAI ChatGPT: What's an interesting topic to discuss for a finance presentation about "LLMs for Language and Beyond"?

Some specific applications of LLMs in finance include:

1. Forecasting stock prices: LLMs can analyze news articles, earnings reports, and other financial data to predict future stock prices with greater accuracy than traditional models.
2. Fraud detection: LLMs can be used to detect patterns of fraud in financial transactions and other data sources, helping to prevent financial losses and improve security.
3. Risk management: LLMs can analyze financial data to identify potential risks and forecast the likelihood of different outcomes, helping investors and financial institutions to make more informed decisions.

Overall, LLMs have the potential to revolutionize the finance industry by providing more accurate and sophisticated predictive models that can help investors and financial institutions to make better decisions and achieve better outcomes.

LLMs such as GPT are phenomenal at recognizing complex patterns in structured and unstructured data

Unique opportunity in Financial Services:

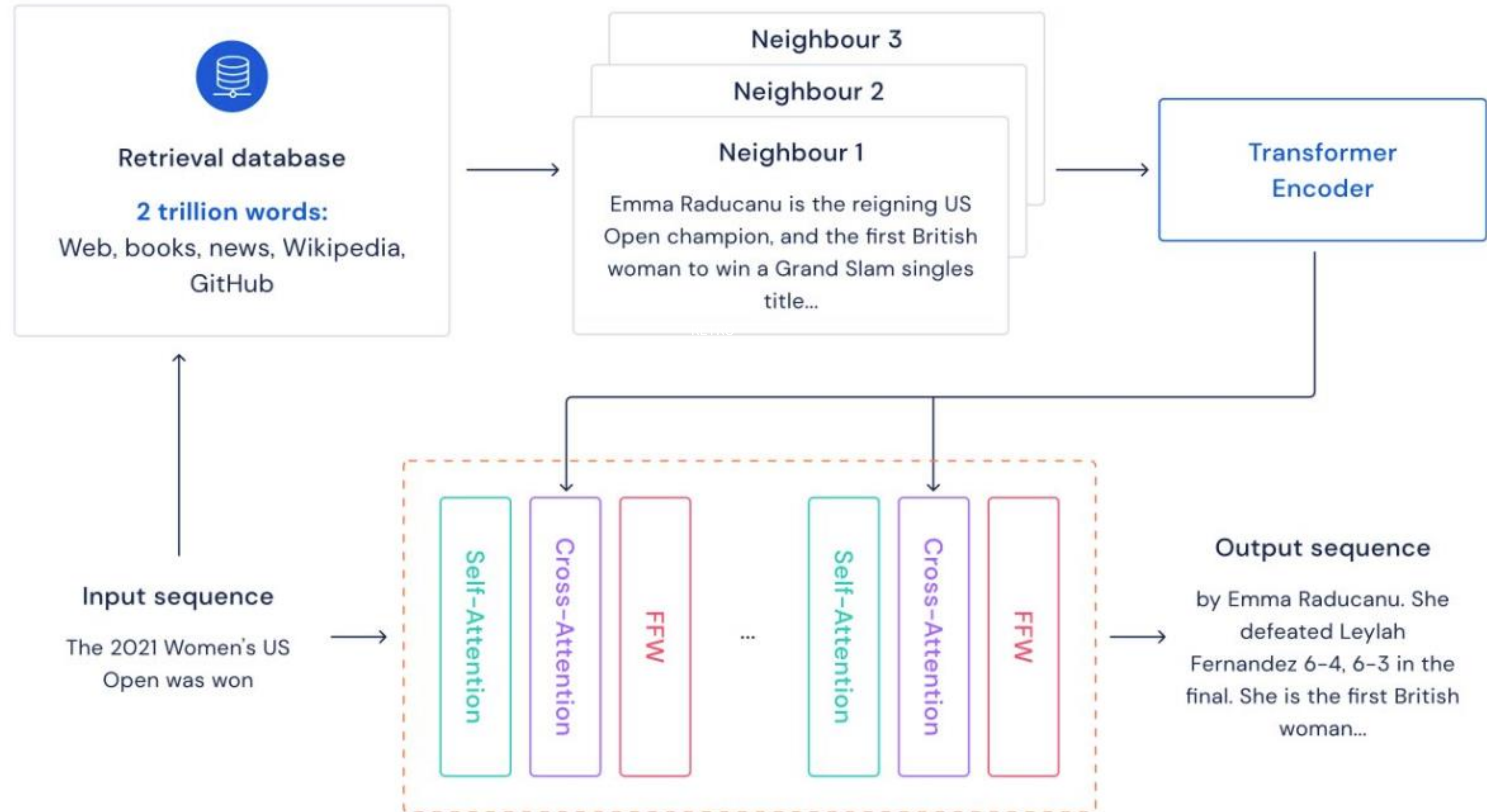
- Chat with References
- LLMs for Synthetic Data Generation and Forecasting



# Generative AI can Chat it up and back it up

LLMs can provide references during chat

Today: DeepMind RETRO - Retrieval Enhanced Transformers



Near Future: ChatGPT + RETRO

for combined chat interface that provides support/references



# LLMs for Synthetic Data Generation and Forecasting

LLMs trained on Structured Data

## Why is this important?

- Necessary for increasing model robustness and accuracy
- Maintain privacy
- Increase data diversity
- Sharing data with external stakeholders

## Important Features:

- **Representative of underlying real data**
  - Real and Synthetic Data have same columns
  - Data are drawn from a similar distribution
  - Synthetic data accurately represents global trends, and local trends in the real data
  - Relevant cross-column categorical features (i.e. city, state)
- **Privacy-focused**
  - Does not leak information about specific entities in the real data
- **Conditionally Generated**
  - Generate new data based on a provided “context”
  - Generate new edge case data



# TABULAR DATA IS STRUCTURED

## CREDIT CARD DATA

Real - 24M rows

user	card	amount	date	year	month	day	hour	minute	use chip	merchant name	merchant city	merchant state	zip	mcc	errors	is fraud
791	1	68.00	2018-01-02 09:10:00	2018	1	2	9	10	Swipe Transaction	12345536	New York	NY	10017	8005	<NA>	0
1572	0	572.42	2018-04-12 07:11:00	2018	4	12	7	11	Chip Transaction	49908535	Princeton	NJ	19406	5634	<NA>	0
2718	7	123.10	2019-01-04 10:14:00	2019	1	4	10	14	Chip Transaction	43211536	Beverly Hills	CA	90210	4800	<NA>	0
21	2	42.04	2020-06-23 11:18:00	2020	6	23	11	18	Swipe Transaction	65423006	Burke	VA	22015	5604	<NA>	0
1001	1	5000.00	2020-11-03 01:22:00	2020	11	3	1	22	Online Transaction	75434546	<NA>	<NA>	<NA>	1234	<NA>	1

Synthetic - 42M rows

user	card	amount	date	year	month	day	hour	minute	use chip	merchant name	merchant city	merchant state	zip	mcc	errors	is_fraud
1010	3	68.64	2019-07-22 12:43:00	2019	7	22	12	43	Chip Transaction	2027553650310142703	Boxford	MA	01921	5541	<NA>	0
142	0	2.21	2004-10-07 06:08:00	2004	10	7	6	8	Swipe Transaction	-6571010470072147219	Seattle	WA	98102	5499	<NA>	0
1037	1	24.32	2014-11-23 17:41:00	2014	11	23	17	41	Swipe Transaction	3959361429988996167	Tucson	AZ	85719	5912	<NA>	0
1734	0	29.60	2004-11-26 22:20:00	2004	11	26	22	20	Swipe Transaction	-4530600671233798827	Menlo Park	CA	94025	5812	<NA>	0
118	1	60.72	2018-11-16 21:53:00	2018	11	16	21	53	Chip Transaction	4751695835751691036	Anaheim	CA	92801	5814	<NA>	0

Additional Benefits include generating:

- Graph data
- Labeled data
- Temporal data
- "Entity" specific data (i.e. customer, or equity)
- Cross-column correlated features



# CHALLENGES OF DIRECTLY APPLYING NLP TOKENIZER TO TABULAR DATA

NLP tokenizer has no table structure information

Government,Canada,Carretera,None,1618.5,20.00,32370.00,32370.00,16185.00,16185.00,1/1/14,January,2014

NLP  
Tokenizer

Government , Canada , C arre tera , None , 16 18 . 5 , 20 . 00 , 323  
70 . 00 , 323 70 . 00 , 16 185 . 00 , 16 185 . 00 , 1 / 1 / 14 ,  
January , 2014



# SOLUTION: SPECIAL TABULAR TOKENIZER

Tokenizer accounts for the table's structural information

Government,Canada,Carretera,None,1618.5,20.00,32370.00,32370.00,16185.00,16185.00,1/1/14,January,2014

Tabular  
Tokenizer

Government

Canada

Carretera

None

1618.5

20.00

3237.00

3237.00

1618.5

1618.5

1/1/14

January

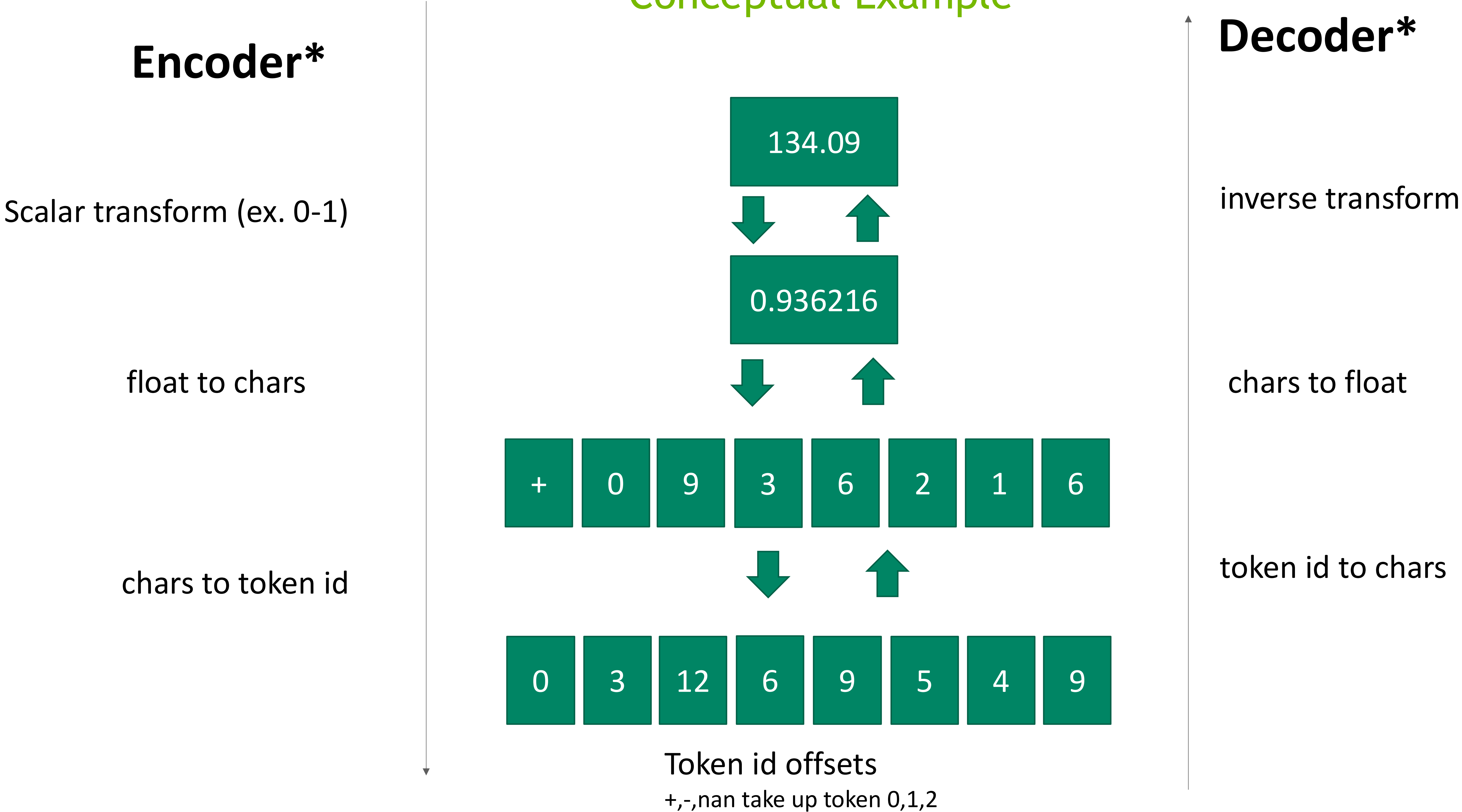
2014



# SPECIAL FLOAT NUMBER ENCODER/DECODER

Encodes numeric value to tokens, and decodes back to numeric value

## Conceptual Example

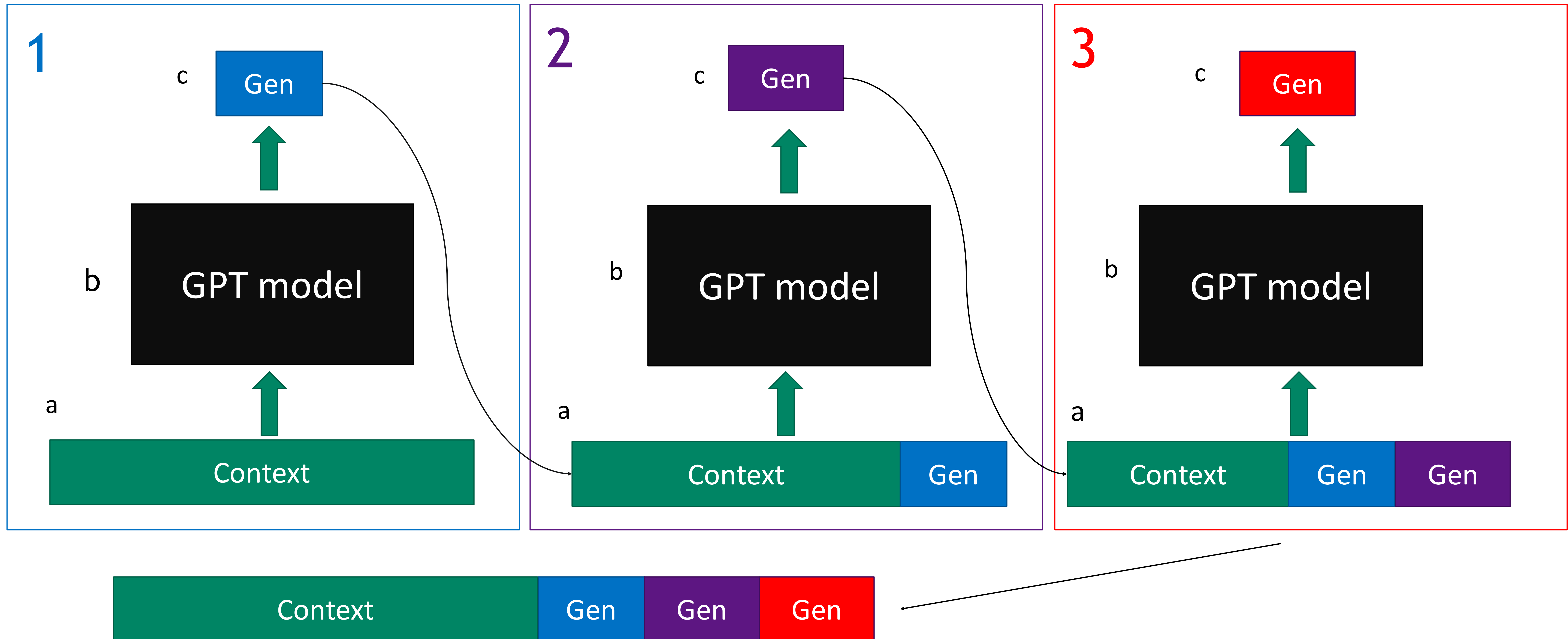


\* column meta data (digits/min/max/transform) precalculated



# CONDITIONAL DATA GENERATION FOR LONG SEQUENCES

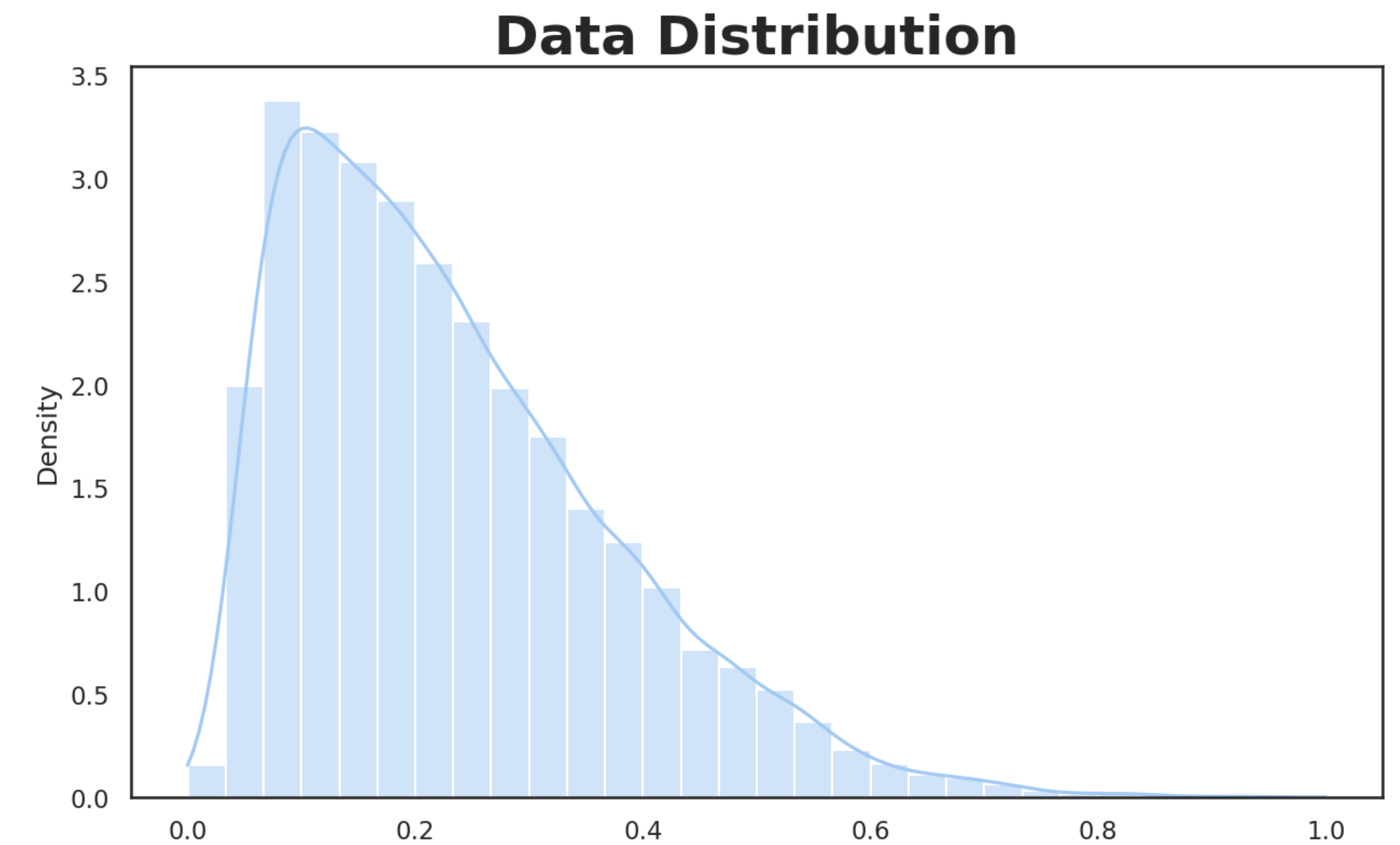
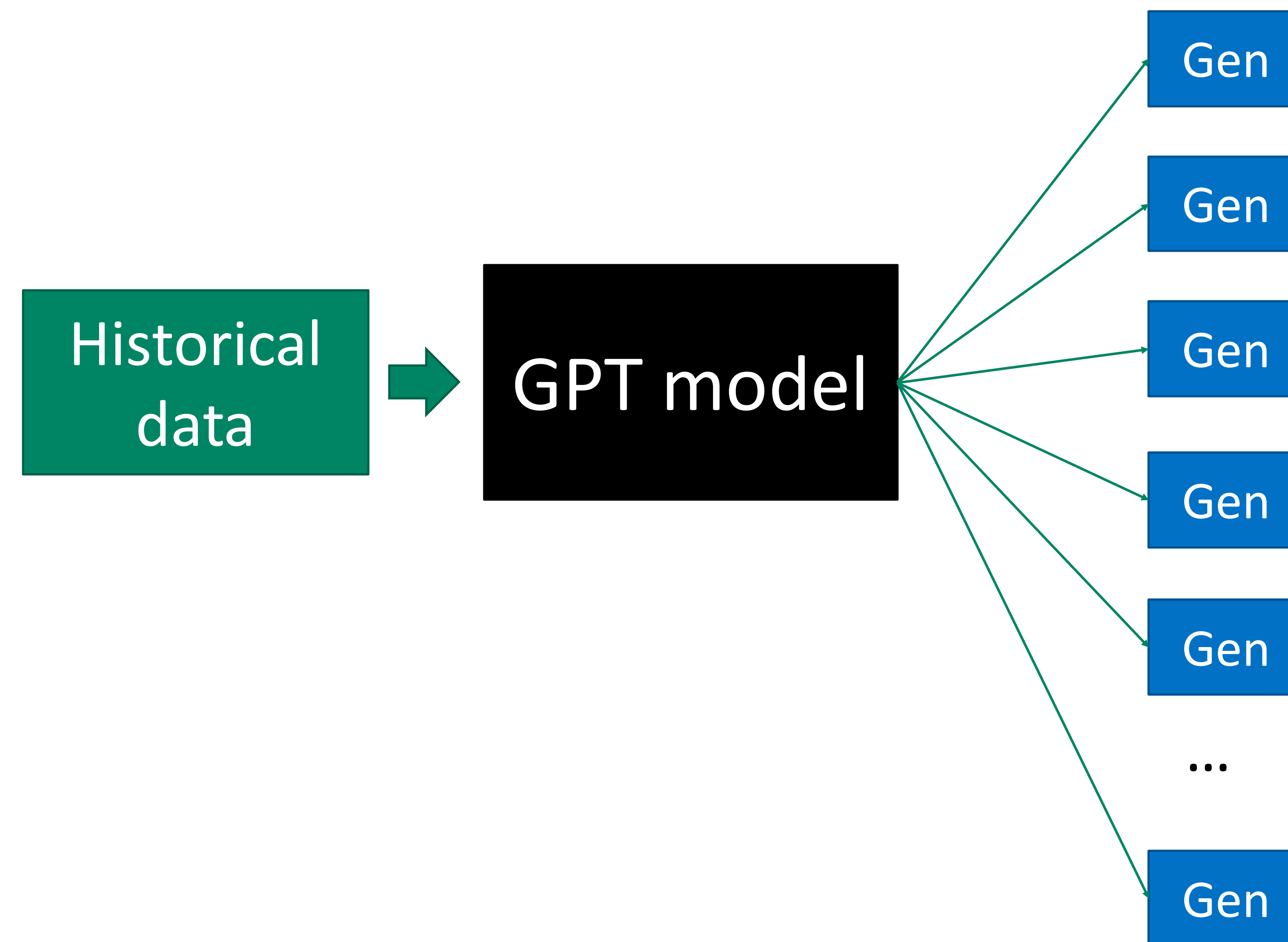
Add newly generated context to previous context





# MULTIPLE SAMPLING TO ESTIMATION DISTRIBUTION

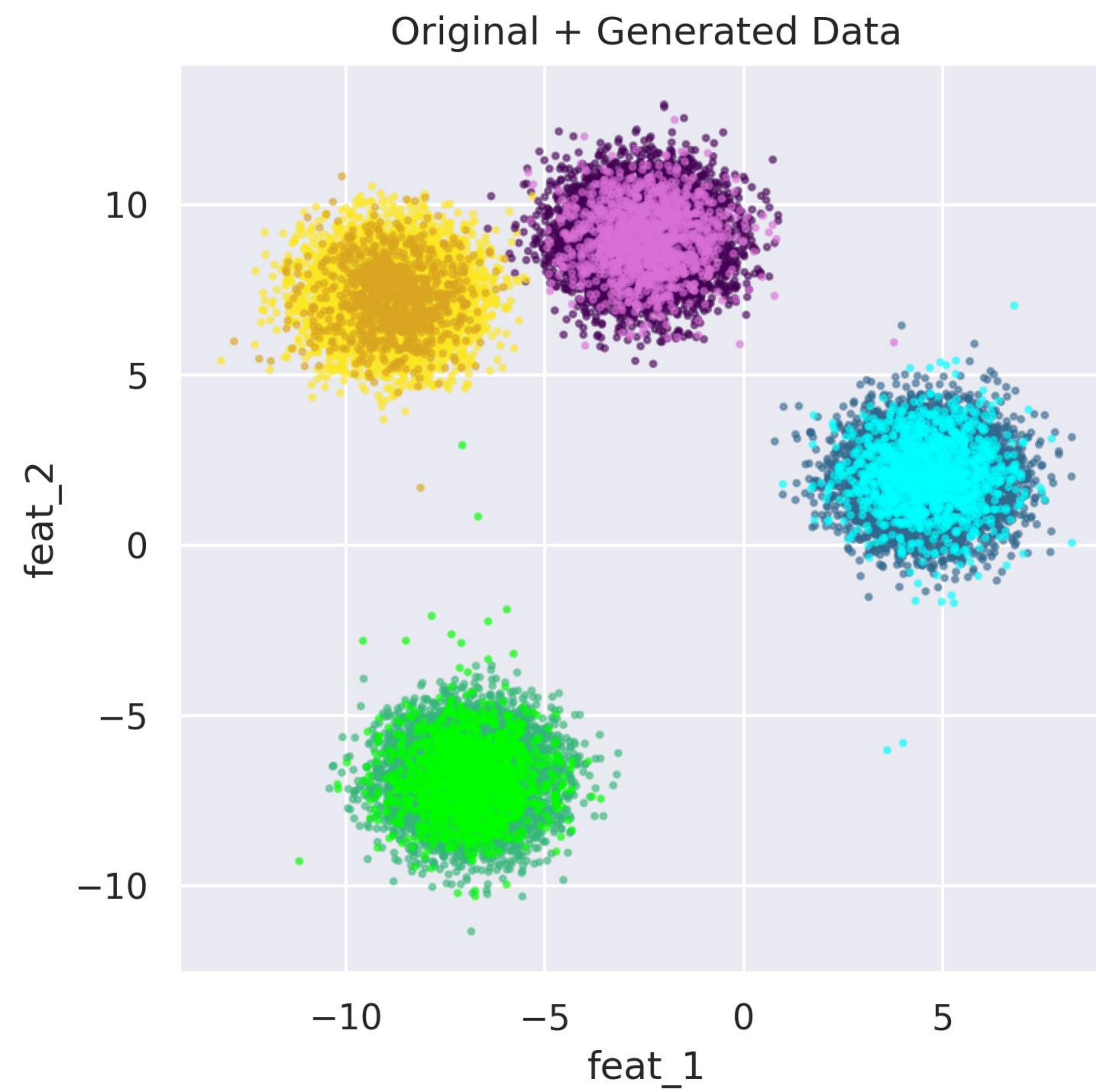
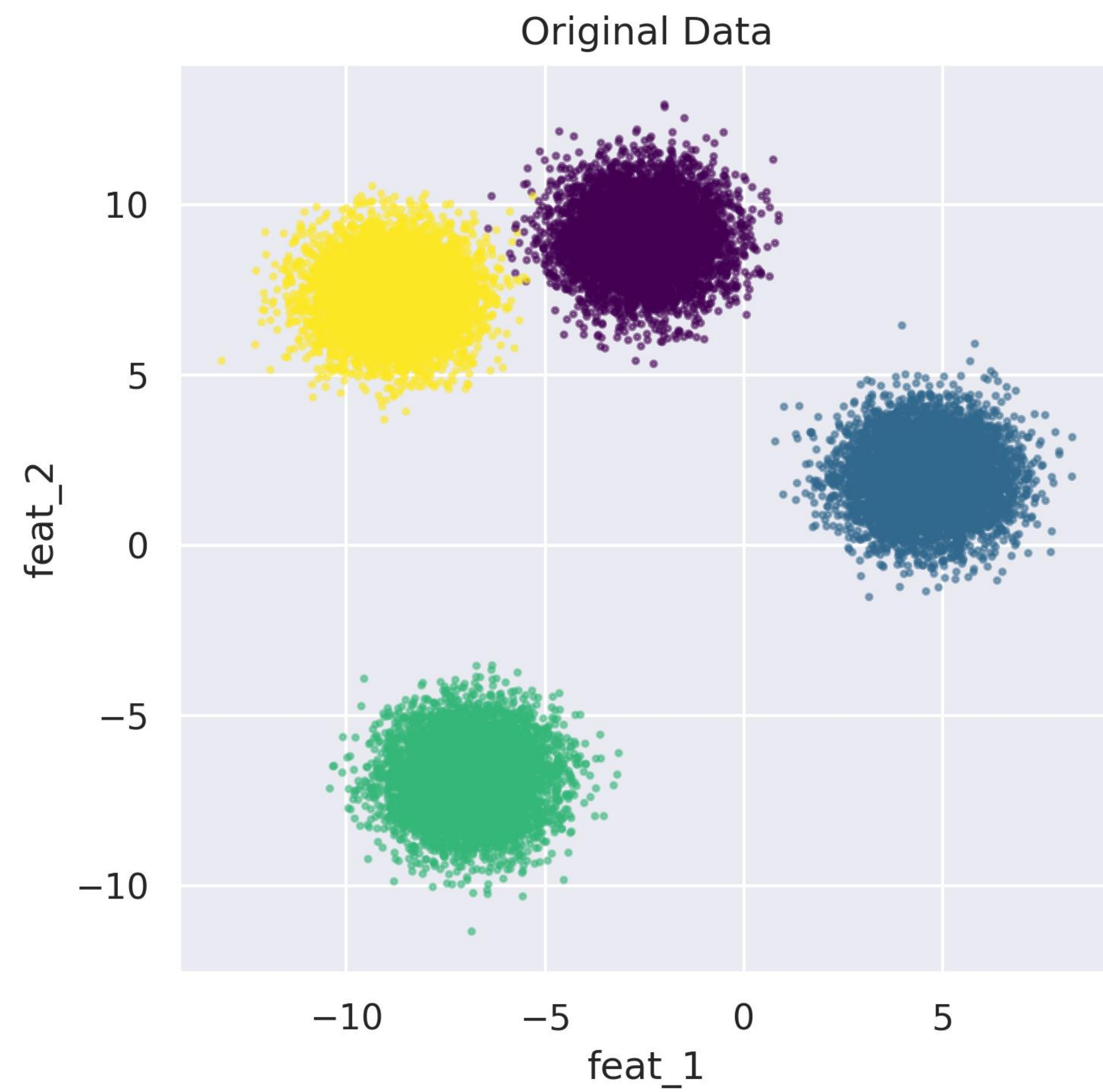
GPT is a stochastic model





# SIMPLE SYNTHETIC DATA GENERATION

Demo





# Conclusion

- LLMs can transform the finance industry by providing **powerful tools** for analyzing and interpreting financial data.
- With their ability to analyze vast amounts of text-based data, LLMs can help investors and financial institutions to **generate insights** and make more **informed decisions**.
- Using **Synthetic Data Generation**, LLMs can be applied to a wide range of financial tasks, including:
  - Equity forecasting,
  - Fraud Detection
  - Risk Management.
- By leveraging the power of LLMs, financial professionals can **gain a competitive edge** in a rapidly changing market, where accurate and timely information is key to success.
- As the field of AI continues to evolve, we can expect to see even more advanced and sophisticated LLMs that will enable **new and innovative applications** in finance and beyond.



