The Fast and the Fabulous. Harnessing GPU Power for High-performance Life Insurance Computations

Karol Maciejewski /
Mehdi Echchelh

Milliman

EAA e-Conference on
Data Science & Data Ethics

16 May 2023

CONTENT OVERVIEW

CASHFLOW AND ALM MODELS

001

CASHFLOW PROJECTIONS AND ALM MODELS

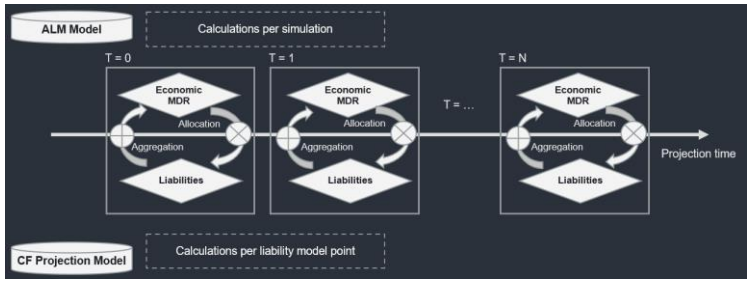
- Cashflow projection models – cornerstone of modern life insurance modeling:

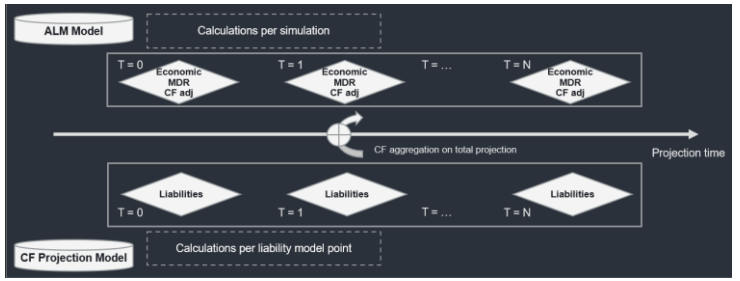
BEL & MV balance sheet projections	Solvency II & capital projections	IFRS 17	Pricing & profit test	Business planning	Valuation and M&A

- Typically, a modern ALM model consists of highly dimensional calculations of:

Liability CF projection	Asset side (ALM)	A ←→ L link
Policy/Model point level	Aggregate (liability) level	Dynamic (full ALM)
Deterministic (expected value)	Stochastic (w.r.t. economic)	Flexing
Decrements, cashflows	Asset and portfolio strategy	
Base reserves (technical/MGR)	Discretionary management rules	

- Full ALM vs Flexing:





PYTHON

010

PYTHON AND ACTUARIAL MODELING

Since its creation, Python has become an important all-purpose language, and its popularity is also increasing among actuaries, notably due to :

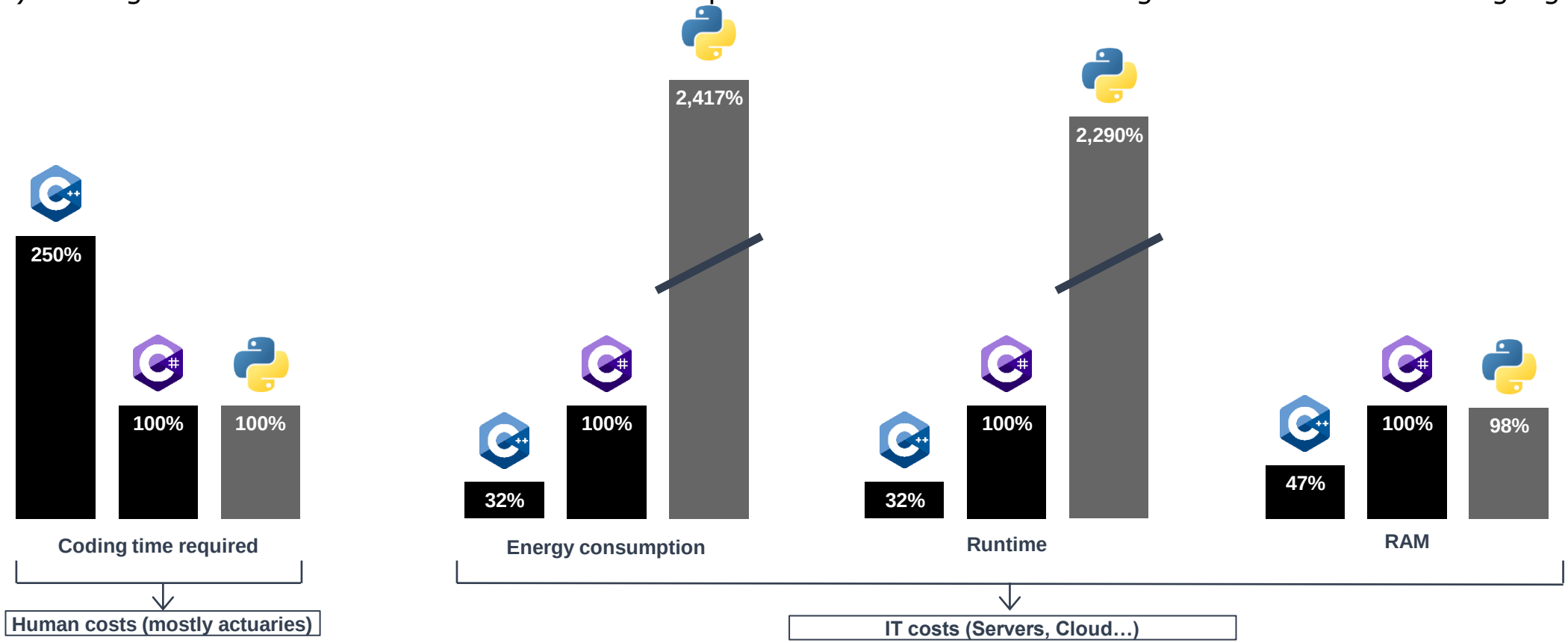
Community	Fast learning curve	Flexibility
• Python is in the top 5 of the most used programming languages(1), and is the most popular language for data science • The language is still evolving to keep the pace with the users needs (e.g., version 3.11 with improved performance)	• Simple syntax : no data typing and a minimalistic and intuitive syntax • Beginner friendly : interactive interpreter (REPL) that allows to experiment with code and get instant feedback	• Many free well-maintained packages for various purpose: Visualisation, Machine Learning, High Performance Computing, Web Development, etc.
Despite some new competition, Python is expected to continue to grow in community and remain used for a long time	Python is currently included in most science related uni curriculum, including **actuarial curriculum**.	More and more packages that can be useful for actuaries. As an example, the CAS has specific open-source python packages for actuarial modelling(2)

➢ We see actuaries using it more and more for various purposes (in non-life and life, for data manipulations and ML workloads, from small ad hoc analyses to production-grade models).

(1) Stack Overflow Developer Survey 2022
(2) Casualty Actuarial Society · GitHub

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

However, Python has the reputation of being a "slow" language (see Ranking Programming Languages by Energy Efficiency (2021) (1)). The figures bellow show benchmarks on the implementation of 10 different algorithms(2) on several languages



(1) Reference paper: R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. Fernandes, J. Saraiva, Ranking Programming Languages by Energy Efficiency (2021) https://haslab.github.io/SAFER/scp21.pdf

(2) Algorithms from *The Computer Language Benchmark Games (CLBG initiative)*

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

- Some advantages of the ease of use of Python can become its flaws with respect to performance

  - Lack of data typing ⇔ huge overhead (esp. on simple arithmetic operations)
  - Pythonic way of coding ⇔ sometimes unintuitive for people using other programming languages
  - Non-pythonic way of coding ⇔ will work too, but almost always will be much slower

- Implementation of the same algorithm in different ways might yield differences in performance in order of 100x

- Difference between "knowing Python" and knowing how to write high-performance Python code
- Writing high-performance Python usually relies on specialized libraries written in lower-level language (typically C), as for example Numpy, Pandas (or new Polars), Numba, PyArrow, …
- Using them, it is possible to overcome most of innate Python performance issues, but there are also limits of what is implemented in each of those "efficient" libraries.

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

```
import dis
dis.dis(add)

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

```
import dis
dis.dis(add)

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

```
TARGET(BINARY_ADD) {

# deal with the string case
…

sum = PyNumber_Add(left,
right)
…

}
```

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

```
import dis
dis.dis(add)

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

```
static PyObject *
long_add(PyLongObject *a,
PyLongObject *b)
PyLongObject z* {

…
z = x_add(a, b)
…

}
```

```
TARGET(L    ARY_ADD) {

# deal with the string case
…

sum = PyNumber_Add(left,
right)
…

}
```

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

```
import dis
dis.dis(add)

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

```
static PyObject *
long_add(PyLongObject *a,
PyLongObject *b)
PyLongObject z* {

…
z = x_add(a, b)
…

}
```

```
TARGET

# deal
…

sum =
right)
…

}
```

```
static PyObject *
x_add(PyLongObject *a,
PyLongObject *b)
PyLongObject z* {

…
…

}
```

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

"Pure" Python implementations are not satisfactory for massive calculations

- Dynamic typing adds overhead costs at runtime. Bellow an example with a simple addition :

```
def add(a, b):
    return a + b

add(1000, 2000)
```

```
import dis
dis.dis(add)

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

```
static PyObject *
long_add(PyLongObject *a,
PyLongObject *b)
PyLongObject z* {

…
z = x_add(a, b)
…

}
```

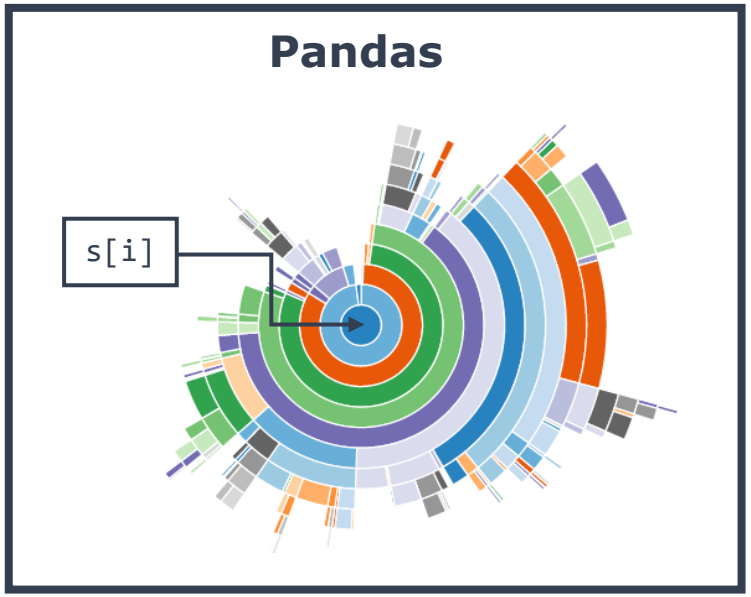
```
static
x_add(Py
PyLongO
PyLongO

…
…

}
```

```
TARGET

# deal
…

sum =
right)
…

}
```

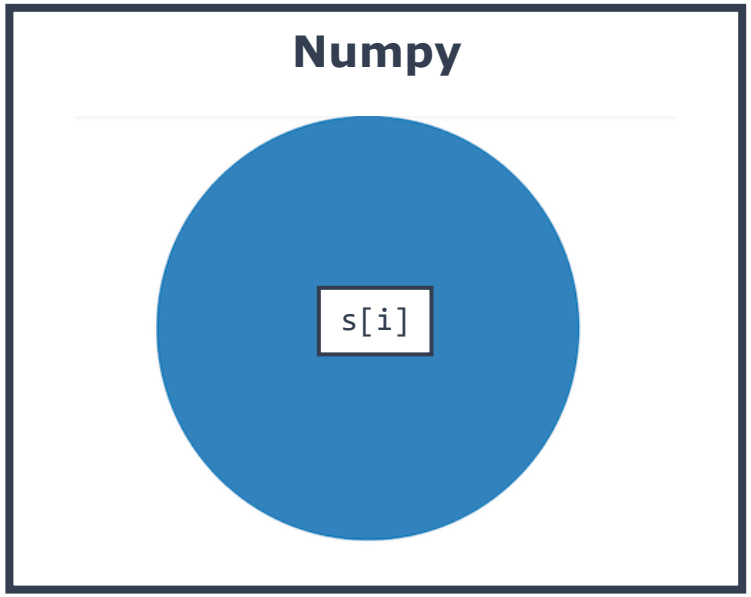
• • •

PYTHON PERFORMANCE: PROBLEMS, REASONS AND SOLUTIONS

Python has specific libraries designed to address those performance issues by using more compiled code and less pure Python. However, the level of pure Python calls executed may still vary depending on the specific library we select, which can then impact performance. The graphs bellow show a comparison in the number of python calls between pandas and numpy for a simple indexing call (s[i]):



- The indexing call is composed of multiple python calls, which yields to additional overhead with pandas.



- The indexing call is directly handled by C++ compiled code, which is more efficient.

➢ Are those differences significant when designing an ALM model?

GPU COMPUTING FUNDAMENTALS

011

GPU AND HPC IN LIFE MODELING

CPU VERSUS GPU

Central Processing Unit (CPU)	Graphical Processing Unit (GPU)
• Versatile calculation engine • Very good at heterogenic computations (MIMD*) • Very good at concurrency (switching tasks) • Very fast single-core computations • Limited number of parallel computational units (e.g. 8 cores / 16 threads) • Handles everything that goes on in a computer • Uses general computer RAM and series of CPU cache memories to store data for calculations	• Originally, designed as a dedicated calculation unit for accelerated graphical computations for computer games (specialized in linear algebra) • Over the years extended to handle more generalized workloads (CUDA, OpenGL) and nowadays widely used for blockchain and cryptocurrency calculations as well as ML and AI workloads • Allows highly parallel homogenic computations (SIMD*), a single household level GPU can have 5000+ cores and productivity cards can have double of that • Individual cores significantly less powerful than the CPU ones (but power in numbers) • Parallel computing benefits and drawbacks (race conditions, synchronization, …) • Uses dedicated vRAM on the GPU card and calculations can be run asynchronously from the CPU

* MIMD – multi-instruction multi-data, SIMD – single-instruction multi-data
See Flynn M. “Very high-speed computing systems”, Proc. of IEEE, 1966

GPU ARCHITECTURE FUNDAMENTALS

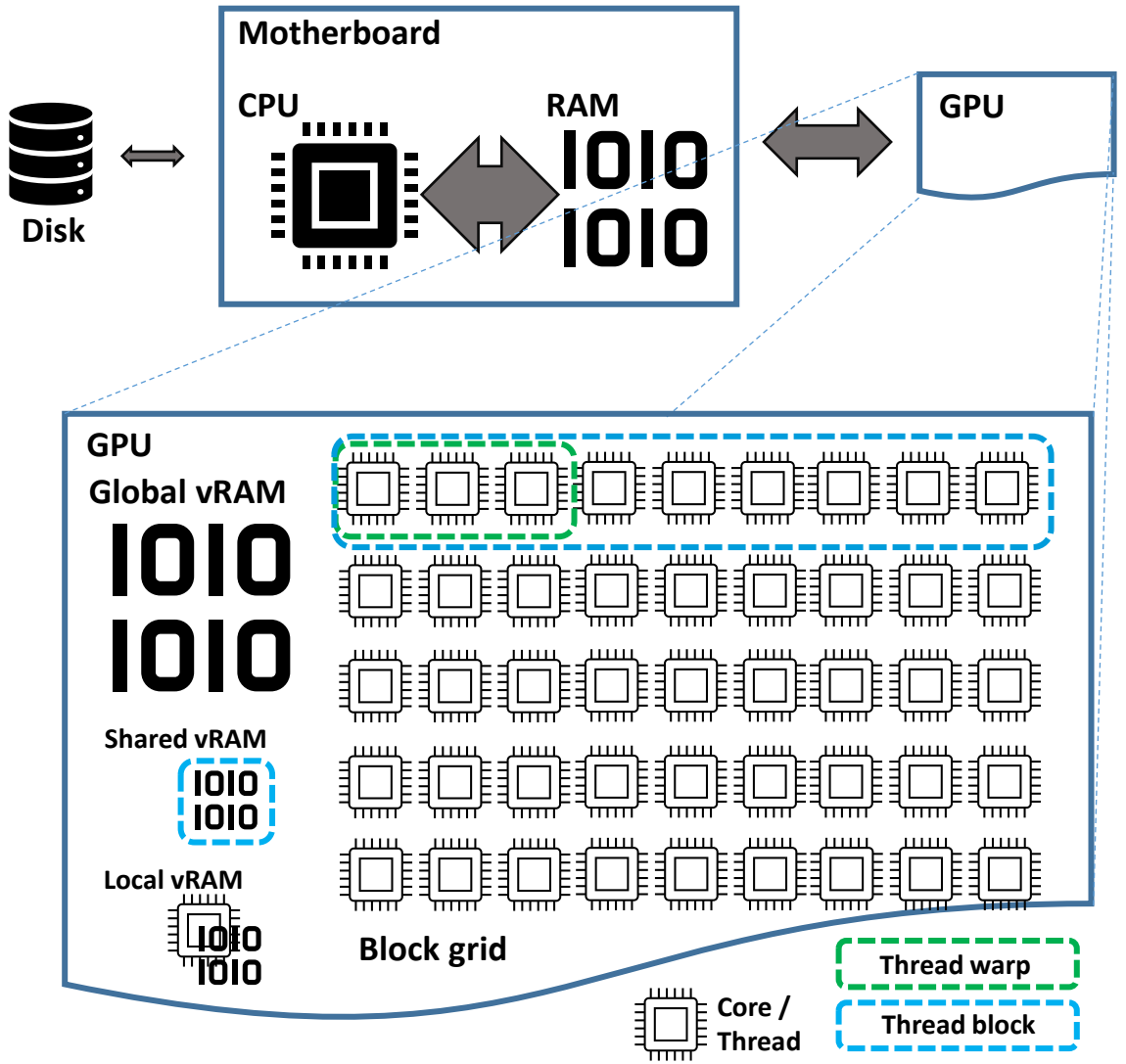
- Typical bottlenecks in data transfers:

  Disk ⇆ **RAM** and also **RAM** ⇆ **vRAM**

- GPU computation grid based on threads organized in blocks, organized in a grid
- Dimensions have both a physical and logical aspect

  Thread ⇆ **Core** **Block** ⇆ **Streaming multiprocessor**

- Dimensionality of indices can be set to match the data problem (1D, 2D, 3D for indexing thread blocks and blocks within the grid)
- Logical calculation grid can be larger than number of cores (each core will be assigned multiple computation cells)
- Number of threads per block → set arbitrarily to a power of 2 (min 32, max 1024)
- Number of blocks → set to match the problem data size (max 2 147 483 647 = 2^31 - 1)

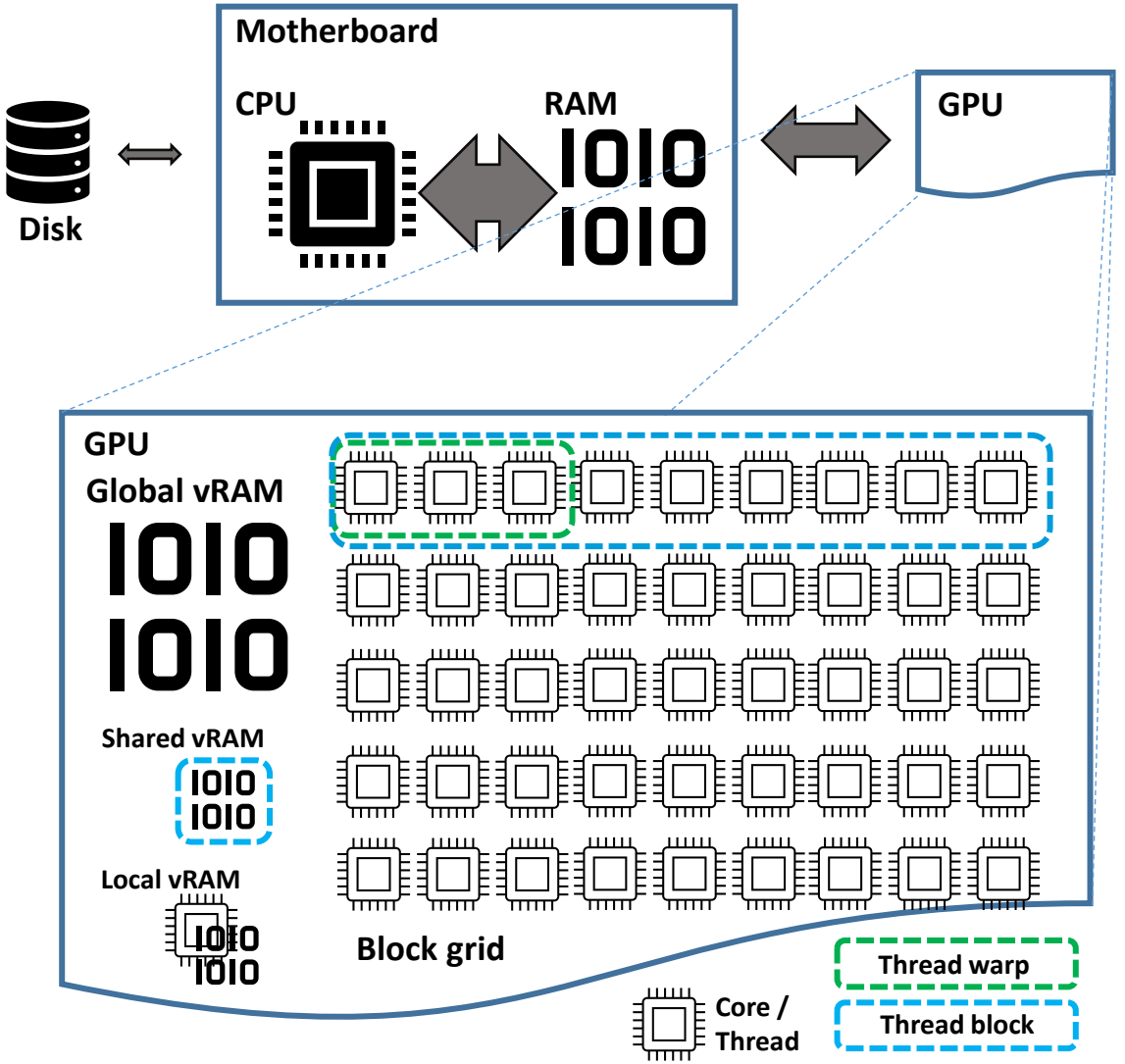


GPU AND HPC IN LIFE MODELING

GPU ARCHITECTURE FUNDAMENTALS

- Choosing number of threads per block:
  - Depends on the type of problem
  - Depends on the required local/shared memory (limited per block/SM)
  - More threads in a block ⇔ less memory per thread
- Number of blocks:
  - [Problem_Size + (N_Threads – 1)] // N_Threads
- For huge data problems other approaches to mapping threads to data can be used (grid-stride approach, single thread loops around data multiples of grid size data cells)

- Tip of the iceberg – optimizing GPU code required deeper understanding of how this architecture works, how it manages and caches memory access, etc.
- For a good example, see Nvidia's presentation on advanced optimizations of a reduction kernel (aggregation): CUDA Webinar 2 (nvidia.com)*



* https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

GPU COMPUTING FUNDAMENTALS

CPU

```python
import numpy as np

def cpu_calc(data):
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            data[i, j] += 1
    return data

data = np.random.rand(10000, 1024)
data = cpu_calc(data)
```

GPU

```python
import numpy as np
from numba import cuda

@cuda.jit
def gpu_calc(data):
    i = cuda.threadIdx.x
    j = cuda.blockIdx.x

    if i*cuda.blockDim.x + j < data.size:
        data[i, j] += 1

data = np.random.rand(10000, 1024)
threads_per_block = 1024
n_blocks = 10000

gpu_calc[n_blocks, threads_per_block](data)
```

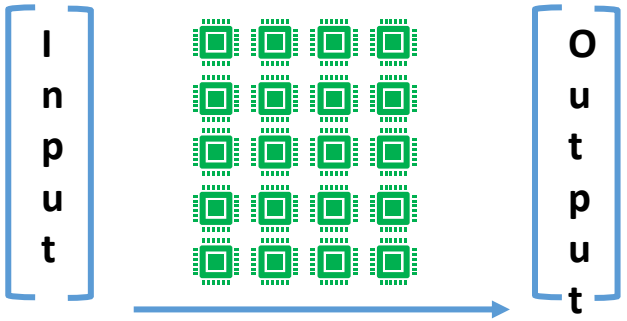
Executed on GPU

The code is for illustrative purposes only – to highlight difference in approach to calculating cells.

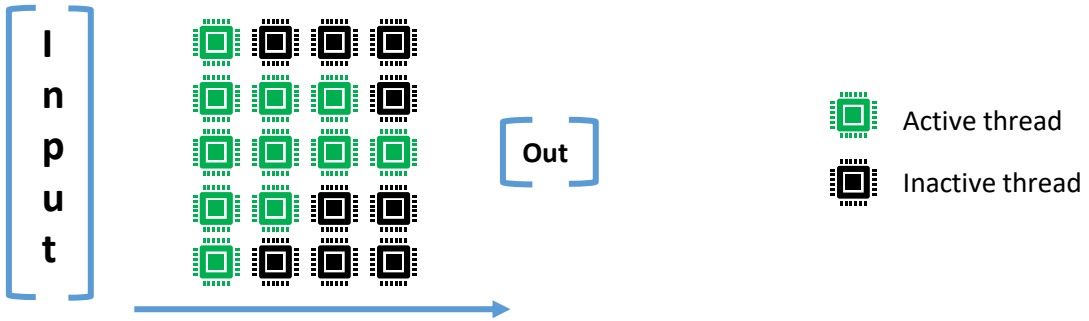
GPU COMPUTING FUNDAMENTALS

Some types of computations and approaches are better suited for GPU than others. In general principles of parallel computing apply, potentially just on bigger scale given the numbers of cores/threads.

- Applying **homogenic** instructions to **large sets** of data to perform **independent** calculations
  - Transfers to / from GPU memory are costly, relative to performing computations on them
  - Better to transfer one large array than many times small chunks of data (but vRAM limitation)
  - Thread synchronization issues might occur if code flow is very not homogenic (e.g. heavy conditionals)
  - Warps of threads have to complete computations together (waiting for last thread of a warp)
  - Race-conditions when multiple threads try to access/write the same cell of data simultaneously
- **Reduction functions** (like aggregation) gain less by running them on GPU, but there are clever ways of optimizing even those types of calculations.



Execution of a regular "map" function on input

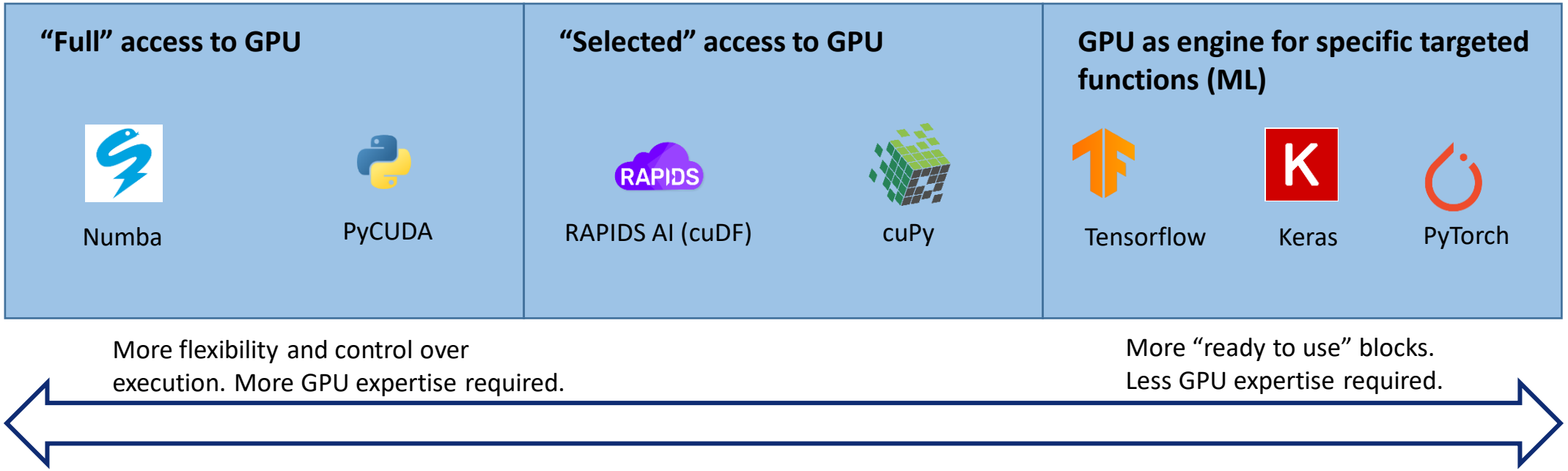


Execution of a "reduction" function on input

GPU COMPUTING FUNDAMENTALS

- Original language for GPU programming: CUDA C++ for Nvidia cards (and later OpenGL for AMD cards)
- Complex, low-level language, very flexible and performant, but high-cost of entry

- In the recent years, this has changed and many libraries facilitating GPU computations are available in Python, for example:



GPU COMPUTING FUNDAMENTALS

CPU	GPU

```python
import numpy as np

data = np.random.rand(100000, dtype=np.float64)
data = np.where(data == 0, 0, np.log(data))
result = data.sum()
```

```python
import cupy as cp

data = cp.random.rand(100000, dtype=cp.float64)
data = cp.where(data == 0, 0, cp.log(data))
result = data.sum()
```

Packages like cuPy or cuDF are drop-in replacements for NumPy and Pandas (or other DF-based library). They are extremely easy to use, if you have your code already using one of the CPU-based libraries.

Note: There are some differences, as there is not 100% compatibility, so not all the code will work directly without any adjustments.

The code is for illustrative purposes only – to highlight difference in approach to calculating cells.

GPU COMPUTING AND LIFE MODELS

Liability cashflow projections

- Independent policy / MP data (100k – several million)
- Similar homogenous calculations for big groups of MP
- Deterministic

- Pure liability CF models offer a perfect case for GPU to shine

Asset models

- Independent stochastic simulations (1000-5000)
- High dimensionality of calculations, so efficient memory use is very important

- Pure asset calculations (like bond MV) are also great case to take advantage of GPU computations

ALM models

- Combines characteristics of the above two, plus:
  - Aggregations of liability and asset data to funds
  - Top-down allocations (profit-sharing)
- Significant differences between flexing and full ALM w.r.t. intensity of those two points

- In some cases (e.g. full ALM) dimensionality multiplies (MP x sims x …), magnifying the problem.
- It is more tricky to efficiently and elegantly implement such model.

GPU AND LIFE MODELS: BENCHMARKS

100

GPU COMPUTING AND LIFE MODELS

1ST BENCHMARK - SIMPLE PROJECTION MODEL

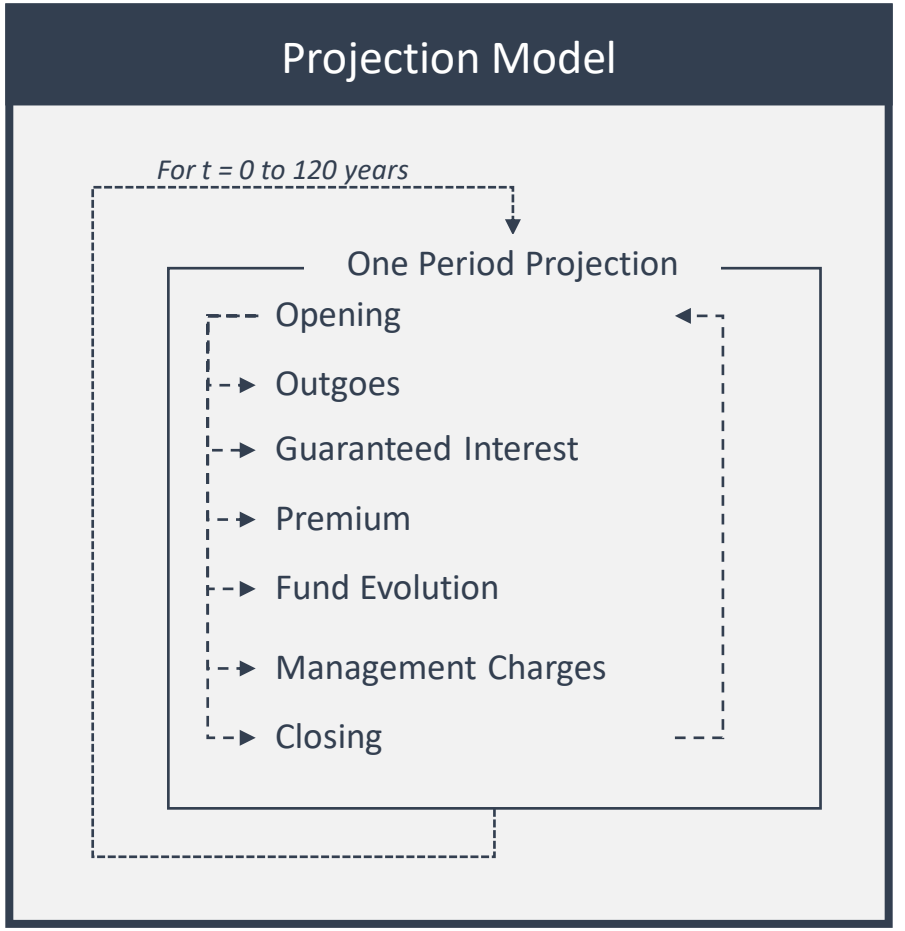
Model implemented

The model developed in this initial benchmark is a liability model for savings product with guarantees and profit sharing. This model projects cash flows at the individual model point level.

- A total of 50 columns are calculated for each model point.
- The number of model points varies, ranging from 10k to 1 Million.

The benchmark highlights the differences in runtime for various implementation methods executed on both CPU and GPU platforms. These approaches consist of:

- Dataframe-based: Implementations utilizing libraries for DataFrame manipulation, such as **pandas**, **dask.dataframe**, and **cuDF**.
- Array-based: Implementations employing libraries for array manipulation, including **numpy** and **cupy**.
- Numba: Implementations based on **numba** just-in-time (jit) compilator for either CPU or GPU (with **numba.cuda**).



GPU COMPUTING AND LIFE MODELS

1ST BENCHMARK - SIMPLE PROJECTION MODEL

Code and Data structures

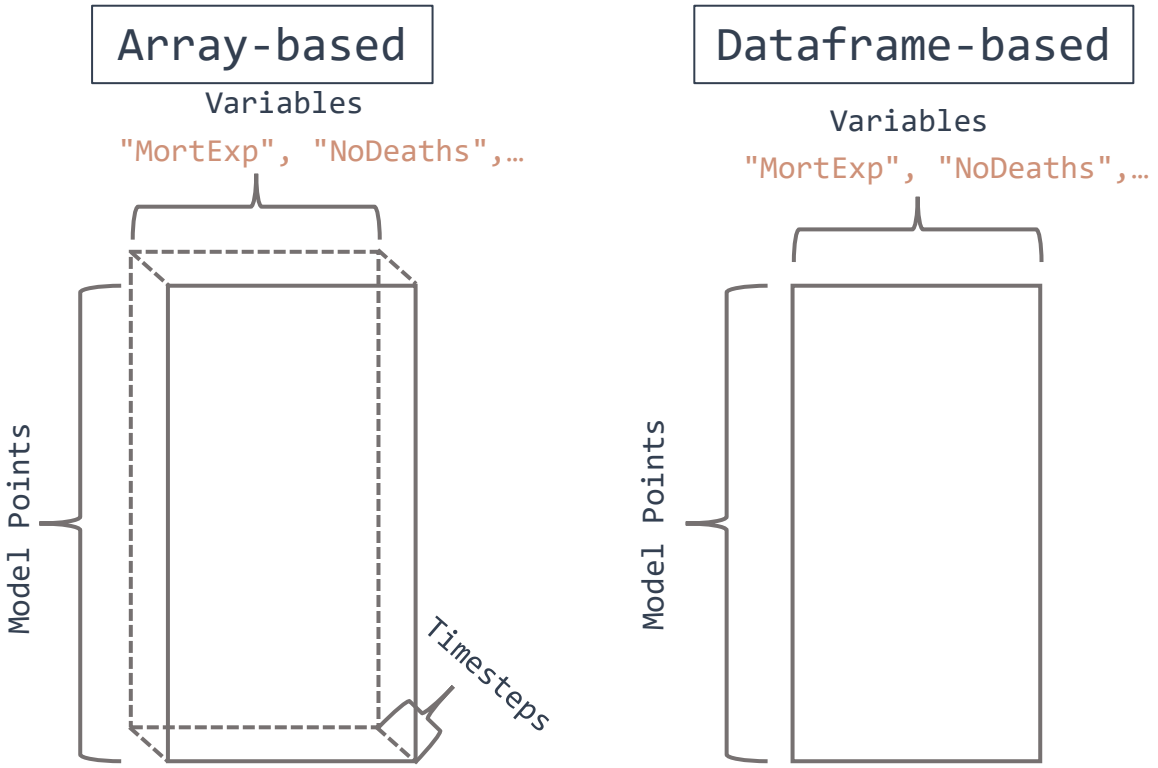
The code architecture was done with two main goals in mind:

- Vectorizing as much calculation as possible (no loop over model points)
- Having a code that is easy to read and to maintain

The data structure used had to take into account the specificities of the libraries used, such as:

- Dask doesn't allow multi-index, so we only considered the model point dimension on the row axis
- CuDf doesn't guarantee the order of the rows after some specific operations (merge, groupby, etc.), but the index stays, so we couldn't rely on automatic aligning of the index.

The schema on the right illustrates how the calculation was represented, taking these points into account, using array-based and dataframe-based approaches.



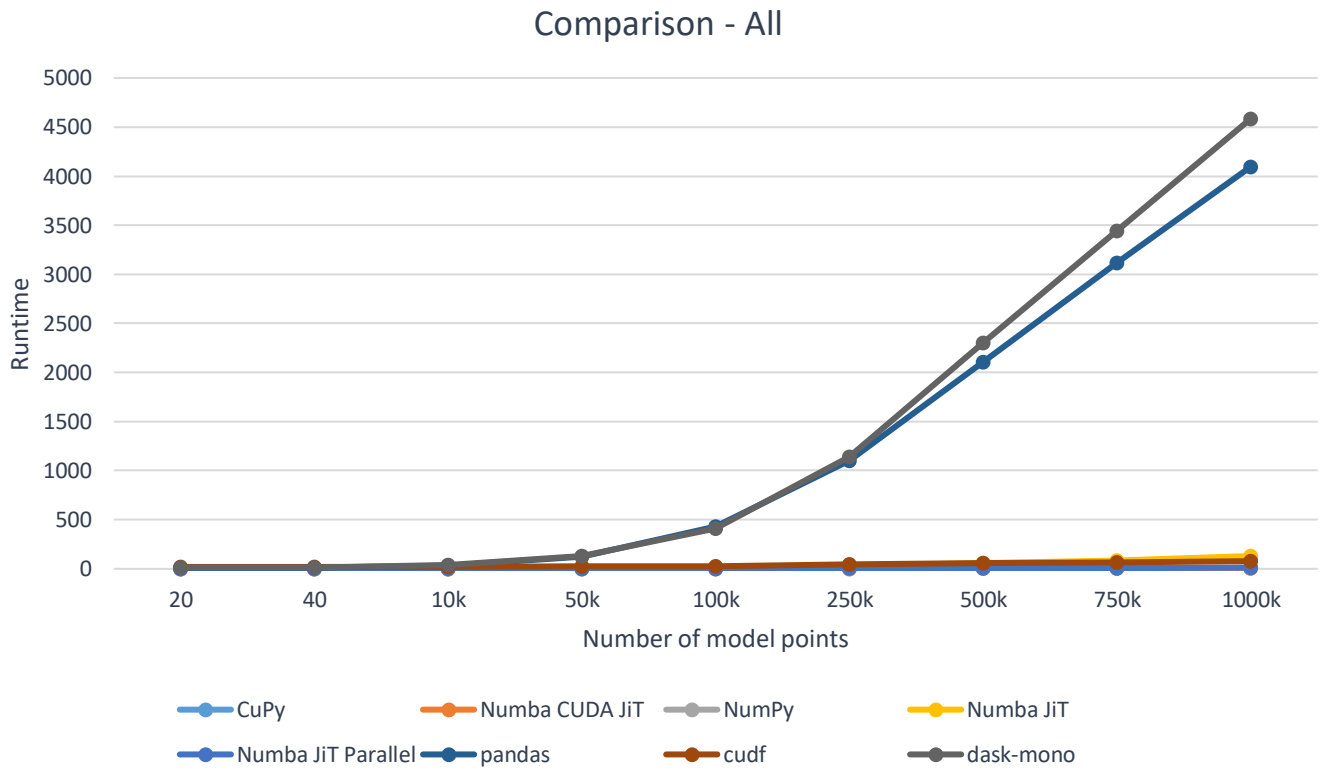
GPU COMPUTING AND LIFE MODELS

1ST BENCHMARK - SIMPLE PROJECTION MODEL

Results

The graph on the right displays the comparison of the runtime for the different implementations (with respect to the number of model points):

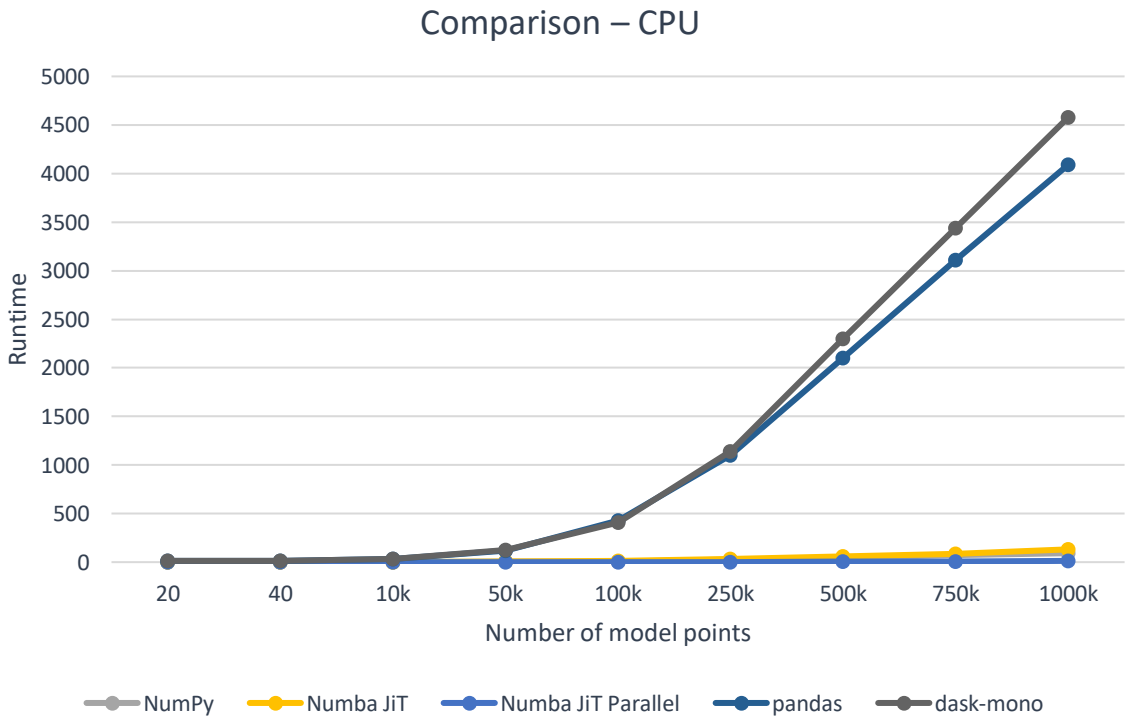
- CPU mono-core
  - NumPy
  - Numba & Numba JiT
  - dask (mono-core)
- CPU multi-core
  - Numba JiT Parallel
- GPU only
  - Cudf
  - CuPy

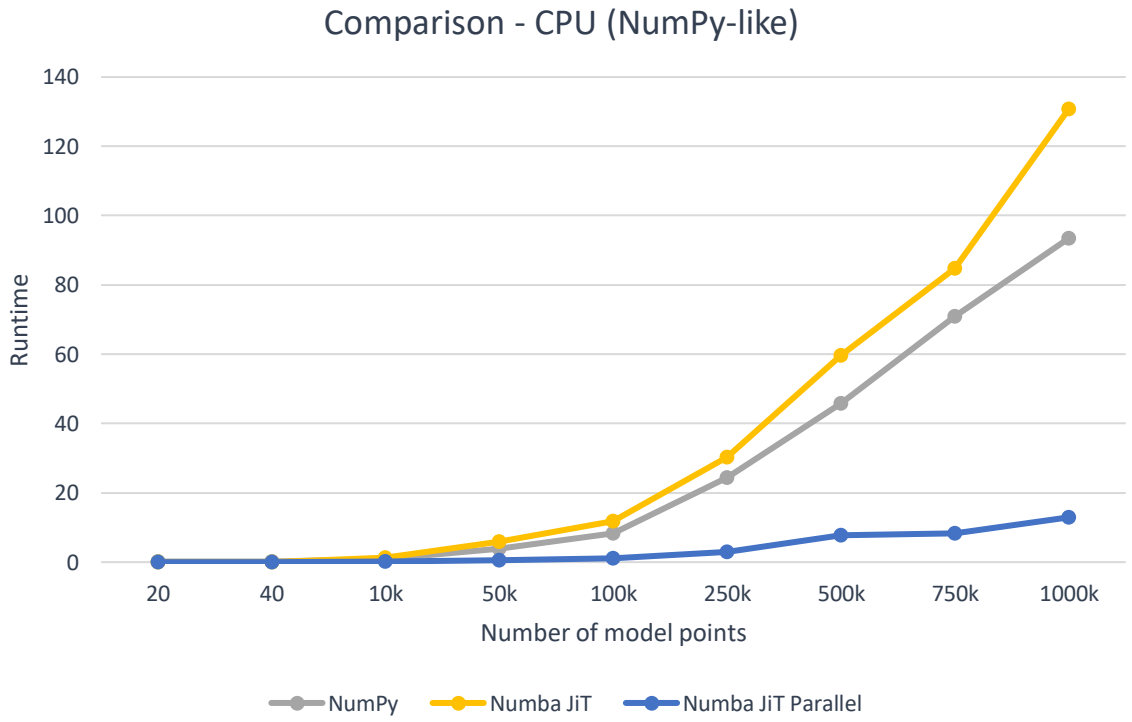


GPU COMPUTING AND LIFE MODELS

1ST BENCHMARK - SIMPLE PROJECTION MODEL

Comparison between CPU-based implementations



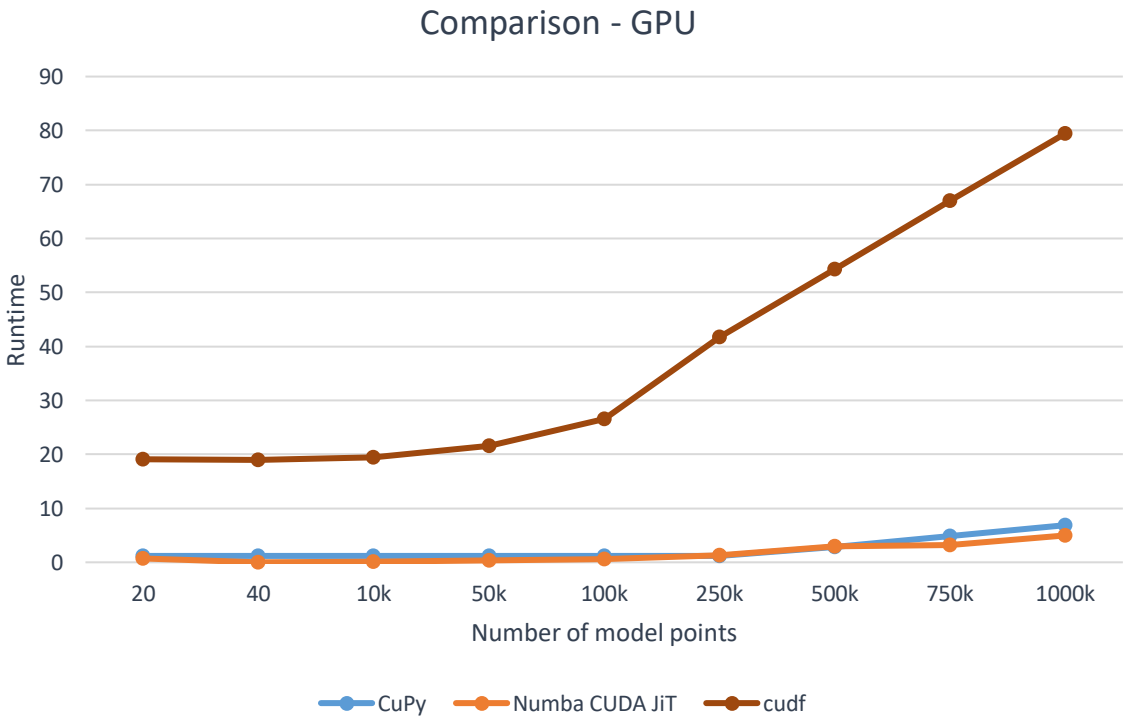


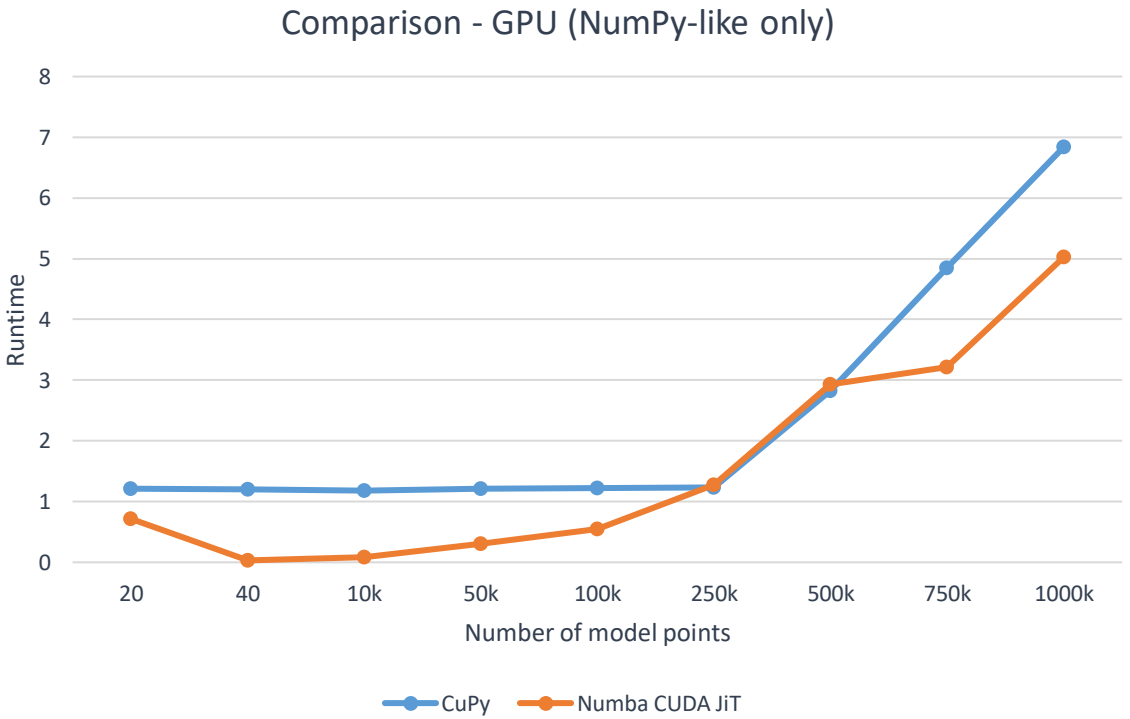
- We see a significant difference in runtime between array-based (numpy-like) and dataframe-based approaches (pandas & dask-mono), with array-based approaches demonstrating superior performance compared to pandas.

GPU COMPUTING AND LIFE MODELS

1ST BENCHMARK - SIMPLE PROJECTION MODEL

Comparison between GPU-based implementations





- On the GPU, the array-based methods (cupy & numba) once again exhibit better performance compared to data-frame approaches (cudf).

GPU COMPUTING AND LIFE MODELS

2ND BENCHMARK – FULL ALM AND FLEXING PROJECTION MODEL

Model implemented

The second benchmark features a more versatile model that can operate in **flexing** mode, with the following features:

- Liabilities: Retirement contract featuring a phase for accumulating funds in both Euros and Unit-Linked investments.
- Assets: modelling of bonds (fixed rate), equities, and cash
- Management rules: target allocation of assets as a percentage of the market values, profit sharing strategy with a target crediting rate, and minimum profit sharing defined by the French regulation.

This model projects cash flows at 4 levels: liability, equity, bond, and pool (which is used for managing the asset-liability interactions).

- Around 175 columns are calculated at the liabilities and pool level and 50 columns at the equity / bond level
- We use the following volumes in the benchmark:

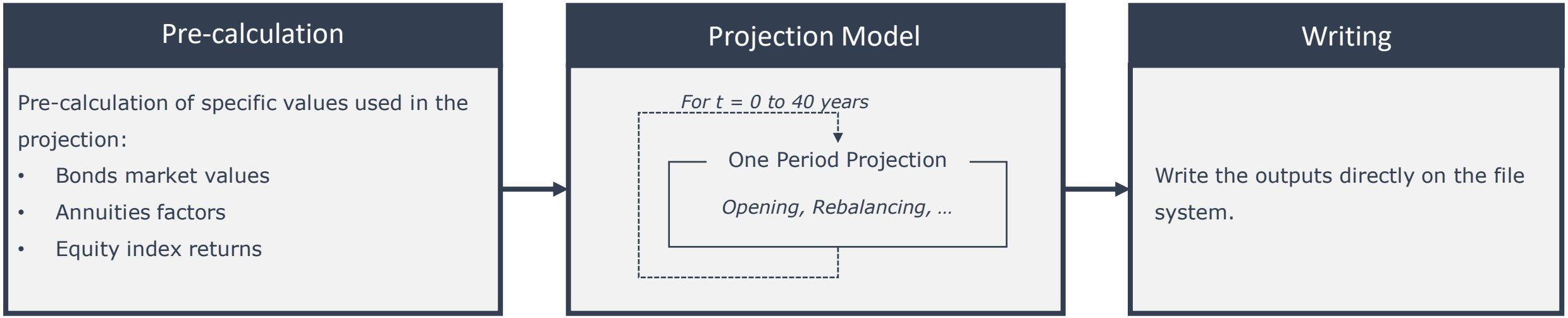
Liabilities	Bond Portfolio	Bond Profiles	Equities	Simulations
1 500	3 000	68	1 400	2 000

GPU COMPUTING AND LIFE MODELS

2ND BENCHMARK – FULL ALM AND FLEXING PROJECTION MODEL

Approach

- The implementation was array-based with numpy / cupy for most of the code and numba for some specific features notably :
  - Recurrence or dependency between model points (e.g., profit sharing cascade, algorithms for selling stocks or bonds)
  - Projection and aggregation of intermediate cash flows (e.g., calculation of bond market values, calculation of annuity factors)
- The implementation was based on a pre-calculation step to try to take advantage of the parallelization on the GPU.

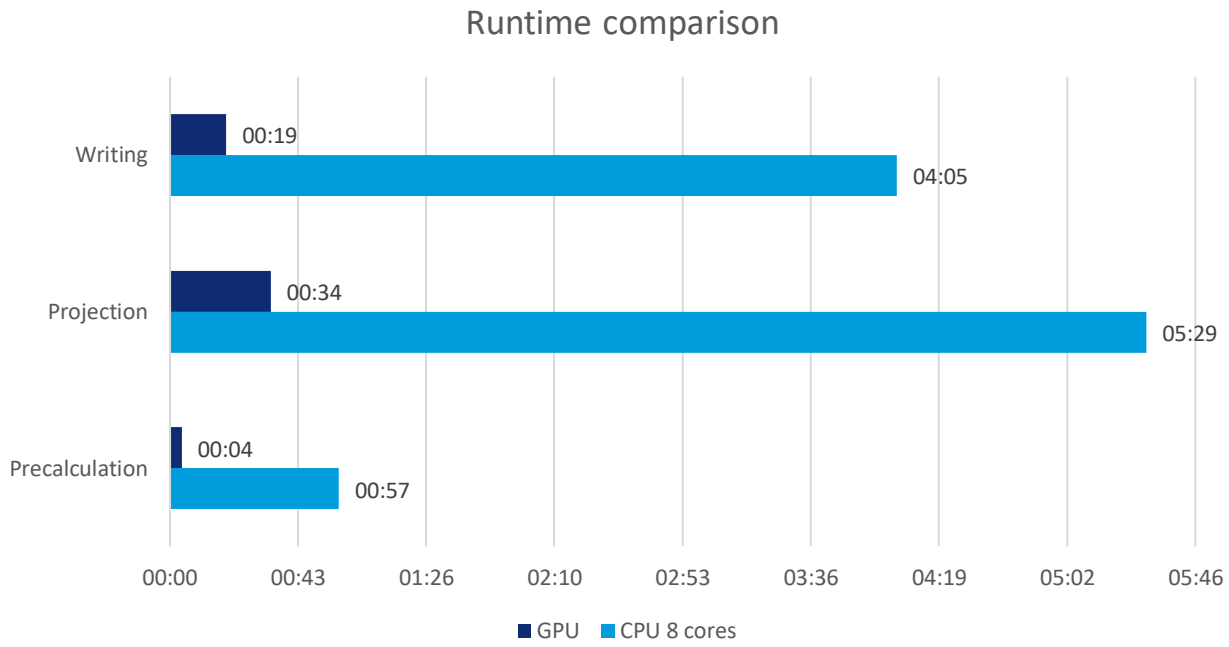


GPU COMPUTING AND LIFE MODELS

2ND BENCHMARK – FULL ALM AND FLEXING PROJECTION MODEL

Results

- The graph on the right compares the runtime between an execution on a single V100 GPU and an 8 core CPU. The CPU execution is distributed across the 8 cores by using Dask.
- The GPU implementation takes advantage of GPU Direct Storage to speed up the writing time.
- As a result, the following speed improvements were observed:
  - The GPU execution is between 10 and 15 times faster for the projection, pre-calculation and writing steps compared to the 8 core CPU.
  - If we compare with a single core implementation, then the GPU is between 80 and 120 times faster than the single core CPU.



GPU COMPUTING AND LIFE MODELS

Key takeaways

ALM models are not like typical ML problems, there are more complex interactions and dimensionality.
Python can be used for numerical HPC (including GPU), but requires extra care and expertise.
GPU has huge potential to speed up certain types of calculations (but not all), e.g.: • Liability CF models • Asset calculations • Flexing ALM approach
It's relatively easy to tap "a little bit" into the GPU potential, but understanding of GPU architecture and technicalities is required to unlock the full potential (even without coding in CUDA C++).
Building HPC models using GPUs and / or clusters is going far beyond typical projection model implementation. It requires combined expertise in actuarial topics as well as IT, and is as much of a model building as a software development endevor.

ABOUT ME

Karol Maciejewski

Milliman

Karol is a director in the Life team of Milliman Paris and a member of the Belgian Institute of Actuaries (IABE).

He has worked combining his actuarial expertise with IT-technical knowledge and passion for technical innovation for life, non-life and data analytics. For the last 10 years, he has been working internationally on ALM and life insurance modeling, focusing on model and process optimization. Most recently, he has been investigating the use of and leading projects using Python and GPU computing for custom high-performance insurance models.

ABOUT ME

Mehdi
Echchelh

Milliman

Mehdi is a consultant in the Data Analytics team of Milliman Paris.

He has experience in the fields of insurance and data science, and is a member of the French Institute of Actuaries (IA). He has a strong interest in computer science and has been actively involved in actuarial modelling topics. In 2021, he joined the analytics team at Milliman Paris, where he focuses on topics such as machine learning applied to insurance, data engineering, and data visualisation.

Thank you very much for your attention

Contact

karol.maciejewski@milliman.com

mehdi.echchelh@milliman.com

EAA e-Conference on
Data Science & Data Ethics

16 May 2023