MILLIMAN WHITE PAPER

Building a high-performance in-house life projection and ALM model: Architecture and implementation considerations in Python

Karol Maciejewski
Mehdi Echchelh
Dominik Sznajder

Milliman

Designing and building a custom life insurance projection and ALM model in-house is a challenging endeavour, which many insurance companies are nowadays considering.

This white paper investigates numerous challenges and decisions that must be faced regarding the architectural choices that fit different functional requirements, as well as implementation approaches that ensure high performance and code simplicity.

Introduction

TEMPTATION OF AN IN-HOUSE MODEL

In recent years, even as proprietary life insurance modelling solutions remain the most common choice, an increasing number of insurance companies have been considering or attempting to move their life cash flow projections and asset-liability management (ALM) models to in-house custom-built platforms.

This might be a tempting alternative to dedicated third-party modelling software. Typically, one or more of the arguments shown in Figure 1 are brought up in favour of such approach.

FIGURE 1: POTENTIAL BENEFITS OF AN IN-HOUSE MODEL

Full customisation to the company needs
Better performance
Better control over costs (e.g., no license fees)
Easier automation and integration with existing processes
More development flexibility
Development of more in-house expertise

In a recently taken survey,[1] most insurance companies indicated that their two main reasons for changing or modernising their modelling solutions was a desire for better automation and industrialisation of the processes, and better model performance. This can certainly be easier and done in a more complete manner, when designing and building a model from scratch using a general purpose programming language, than trying to interface with proprietary systems that were not designed with such purpose in mind. However, some proprietary modelling platforms have been putting significant effort in those areas too.

Apart from that aspect, the majority of proprietary modelling platforms come with some constraints, which in most cases might simplify the decisions required during model development. But they also regularly become the source of simplifications or out-of-model adjustments. These items add up over time and eventually increase the complexity of the model and processes. Custom solutions have the potential to alleviate these problems.

RISKS INVOLVED

While these arguments can be true, there are also numerous challenges linked to both the design and implementation phases, as well as maintenance afterwards. Just as this paper does not intend to be a comparison of proprietary versus in-house modelling solutions, it does not provide a full analysis of the risks involved, as they consist of complex tasks and should be done case by case. We will list a few of the most obvious ones, but it should be kept in mind that there are numerous complexities and issues that have to be tackled in order to build a successful in-house model efficiently integrated into existing modelling and reporting processes.

Full customisation possibly might end up meaning piling up requirements from different stakeholders to include all the ”nice to haves.” Consequently, it might increase the cost of development and the complexity of the model.

[1] Maciejewski, K., Miehe, P., & Peplow, T. (February 2023). Life Insurance Modelling Platforms: Changing Landscape. Milliman White Paper. Retrieved 22 February 2023 from https://www.milliman.com/en/insight/life-insurance-modeling-platforms-changing-landscape.

Better performance is sometimes difficult to achieve, depending on the platform used for implementation and detailed functional requirements of the model. If wrong choices are made in the design phase, the consequences will only be visible later on in the development phase. Making changes to the architecture or functionalities then will cause delays and require additional budget. In general, the cost of development and maintenance can easily be (initially) underestimated. Building a projection model, especially integrated with other elements of the modelling and reporting process, is as much of an IT project as it is an actuarial model. Even with an appropriate architecture, developing performant code requires constant focus on efficiency and economic use of resources, especially in a language like Python, which we use later on as example.

There are additional considerations that are usually not present when using a third-party modelling software. Using a general programming language and doing developments in-house might cause or complicate discussions concerning ownership of the model between the actuarial modelling and IT teams. It also might create resource problems, as such models will require a very specific blend of programming and actuarial skills to be efficiently developed and properly maintained.

There is a balance between high-performance code and code that is intuitive and easy to follow by modelling actuaries. A custom projection model is also to some extent becoming part of the software of the company. Its development process and practices should therefore follow the same principles and rigour as other IT developments.

Going down this path also means getting rid of all kinds of the aforementioned constraints that come with the third-party solutions. There is a much greater degree of freedom of choice regarding, among others, those shown in Figure 2.

FIGURE 2: ADDITIONAL CHOICES IN IN-HOUSE MODEL DESIGN

IT hardware platform
Language or software platform
Functionalities implemented
Architecture of the model
Technologies and libraries used

However, what at first seems like a blessing might quickly turn into a curse, given the number of decisions that must be made. This paper will attempt to give an overview of these choices and provide information about strong and weak points of different options, which can help in making decisions in each individual case.

Architectural and functional choices in cash flow projection models

CASH FLOW PROJECTION AND ALM MODELS

The focus of this paper is on life insurance cash flow projection and ALM models, which are a cornerstone of most life actuarial calculations and reporting, including regulatory, risk management and financial reporting—e.g., Solvency II, local GAAP, International Financial Reporting Standards (IFRS) 17—and pricing and business planning, as well as portfolio valuations. While there are several elements influencing the expected cash flows, the element underlying all projection models is the liability cash flow projection.

Liability cash flow models

As the name suggests, liability cash flow projection models generate expected cash flows of portfolios of insurance liabilities. Based on a set of best estimate assumptions, the expected evolution of the policies' states is calculated and the corresponding sets of cash flows to and from the policies, leading to a complete expected future profit and loss (P&L) in each projection period. Depending on the purpose of the calculations, projection models often use additional sets of assumptions, e.g., valuation assumptions for statutory reserves calculations, pricing assumptions for premium calculations etc. Usually, a liability projection model will generate its output for each individual policy (or model point, as we explain later) at each future calculation time step.

Asset-liability models

Nowadays, in the majority of cases, the projection model comprises both a liability projection engine and an asset model, which computes asset portfolio evolution (market and accounting values and returns). This allows the complete model to include, on one hand, different portfolio management strategies and evaluations of their impact on the results and, on the other hand, dynamic effects of the market conditions and asset portfolio performance on the liabilities, such as profit sharing, dynamic lapses and guarantees or market-specific additional reserves. By making such calculations over a set of different evolutions of the current market conditions, insurance companies can directly evaluate the time value of any embedded asymmetric portfolio behaviours, most often referred to as the time value of options and guarantees (TVOG). This typically means running the model between 1,000 and 5,000 times with different stochastic economic scenarios.

An alternative approach that some companies use involves a closed-form solution derived from option pricing that allows companies to approximate the TVOG after calibration to the current market situation. Using this approach requires a single calculation instead of taking an average of thousands of stochastic simulations, but it yields best results when options and guarantees are based on benchmarks directly observable in the market, for which calibration parameters are available.

MODEL ARCHITECTURE AND FUNCTIONALITIES

There are many choices related to the functionalities of the model. Some are independent of the software platform chosen, and present in any projection model implementation, but there are even more if the path of a custom in-house model has been taken. Out of all those, there are a few key decisions that will have substantial impact on the architecture of the model, its computational complexity and hardware requirements to maintain sufficient performance.

Timing

Most life insurance products are long-term. Typically, the liability cash flows are projected for 40 to 50 years and, in the case of some traditional products involving whole-life or annuities, it can be even up to 100 years. As actuaries are known for their love of accuracy and precision, a vast majority of models use monthly projection steps. In the case of most modern products, however, an annual calculation step provides a good approximation, and it is possible to include a timing adjustment on annual cash flows if necessary. A notable exception is modelling any kind of disability or protection products, where the recovery assumptions usually exhibit significant variability in the first months of the coverage period.

The asset side of the ALM models, on the contrary, has been usually modelled using an annual step. This is linked, on one hand, to the fact that usually major management decision rules and portfolio management rules are determined less frequently than monthly. On the other hand, it is also related to the frequency of economic scenarios that need to be provided. Having to generate all stochastic simulations monthly would in the past significantly increase the computational requirements. However, as computational capacity increases, in some markets, like notably in the UK, the more sophisticated models are actually performing their asset-side calculations monthly as well. This allows them, among other capabilities, to more accurately model the hedging programmes.

An additional aspect of time in the model is intra-annual calculations. This bears most consequence for annual models, when the start of the projection is required at a non-annual point, e.g., at each quarter as in the case of quarterly regulatory valuation. While this does not impact model performance or memory requirements in a significant way, it creates additional complexity in the code.

A well-designed model could allow a flexible time step, with a variable step in the first projection year to allow the intra-annual functionality. This way, the model could be used to periodically show little difference in the results between monthly and annual liability calculation and use the annual step in the usual reporting calculations to speed up the calculation process and limit the running cost.

Model point granularity

Originally, liability cash flow projections have been performed on the individual policy level, as the key drivers for the cash flow patterns are determined by policy and policyholder characteristics. Over the years, and especially in the recent years, several factors have been driving a search for alternatives, in order to group the liability calculations. Three of the main factors include:

- Evolution of liability models into asset-liability models with a dynamic interaction between the two sides of the balance sheet
- Increasing computational complexity coming from requirements of the new reporting frameworks
- Ongoing consolidation of the insurance market, which leads to portfolios with millions of policies

There are two main approaches to reducing the number of input data lines to the model (called *model points*) and, thus, the amount of calculations required to project cash flows. The first one is grouping the policies by the similarity of their characteristics, such as policy term, policyholder age, sum assured etc. This approach is conceptually intuitive and fairly easy computationally, but also quickly reaches its limits. More sophisticated approaches are based on machine learning (ML) methods, such as clustering, which focus on finding similarities not only in policy characteristics, but mainly in projection results.

With good clustering implementation, it is possible to reach even 99% compression rates (one model point remaining after grouping out of the initial 100) with almost perfect results replication. This has significant implications for not only model performance but, in the case of a full dynamic ALM model, sometimes even for the feasibility of running the model. As we will show in the subsequent sections, a dynamic ALM model has rather strict restrictions on architecture and high memory requirements. Reducing the number of policies by a factor of 100 might eliminate the need for much more expensive hardware.

Dynamic ALM or flexing

There are two commonly used methods for taking into account the interaction between asset and liability modules in the model.

In the first approach, there is a dynamic link between liability and asset calculations. For each projection period, first the liabilities are calculated and then the aggregated results are fed into the assets model. The results of the asset portfolio, profit sharing and management actions are then directly included in the liability projection for the next periods, immediately affecting the future evolution.

In the second approach, the liabilities are first projected over the whole projection period, using a base economic situation evolution, e.g., using the technical or guarantee rates for reserve revalorisation and no profit sharing. Afterwards these reserves and cash flows are modified based on the asset model results using multiplicative flexing factors. This happens on an aggregated liability basis, where each group has a similar sensitivity to economic circumstances or asset return changes.

While the first approach is generally considered more accurate, it has much higher computational complexity and stricter architecture requirements. It requires all liability policies to be kept in memory while performing asset calculations (or using an intermediate results cache, which usually drastically deteriorates performance). This not only substantially increases the memory requirement of the hardware platform running the model, but also introduces dependency between liability policies, because of the aggregation step, which limits the potential for parallelisation. In a basic case, the calculations at each time step follow the loop of:

- Individual model point calculations
- Aggregation of the results to liability segments
- Economic evolution of individual assets
- Asset-liability and management decision rules calculations on the asset pool level
- Allocation of results back to model point

For many liability products the flexing approach can, in fact, produce the same results, provided that the flexing formulae are properly determined and calibrated. There is a clear performance and architectural benefit, however, as the liability model can be run deterministically only once for many different economic scenario evolutions—whether it is stochastic simulations of the base run or different market shock scenarios. It also means that each model point can be calculated independently of the other policies and, once calculated, it can be removed from memory.

FIGURE 3: CALCULATION FLOW IN DYNAMIC ALM MODEL

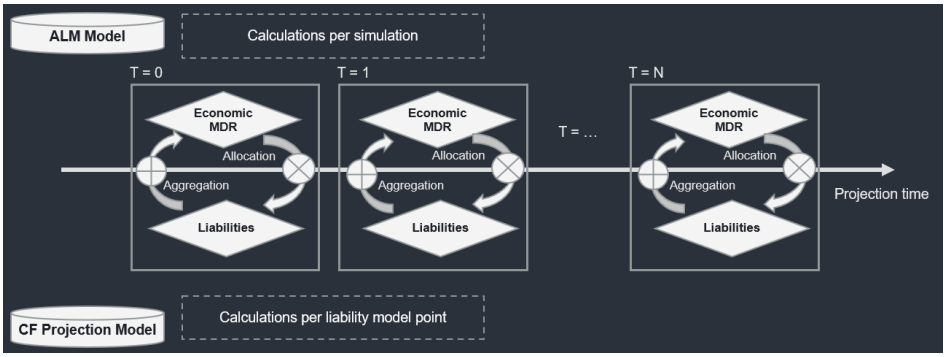
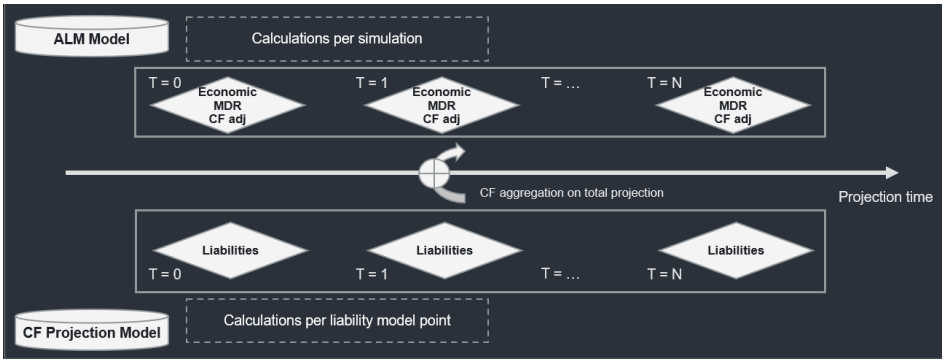


FIGURE 4: CALCULATION FLOW IN FLEXING ALM MODEL



Multi-market or market-specific

Yet another functional and organisational decision with potential impact on the architecture and complexity of the model concerns multi-market companies and the decision whether there should be a single multi-market model or several market-specific ones.

While from a group entity and maintenance perspective, it might be very tempting to have "one model to rule them all," there is also a complexity cost to it, especially if some markets have very specific requirements. Having local models usually allows them to be as simple as possible, only focussing on the requirements of the given market. Trying to incorporate all the local functionalities in a centralised model will likely introduce some additional performance and parameter overhead and might create interactions between functionalities that are not known in any individual market.

The model maintenance process and enforcing group modelling guidelines and standards might be easier, but the code complexity will be higher and local users might find the model more difficult to understand and parametrise.

The need for speed

With the evolution of the insurance industry and development of new reporting standards like Solvency II and IFRS17, there is an increasing demand for more model runs and more features to be included in the projection models. There are also other factors at play. The supervisory agencies tend to ask for more sophisticated modelling approaches for different components of the insurance contracts, assets in the portfolio or management decisions. The competition in the market drives companies to include more options and features in the insurance products. The emergence of new risks in the world, such as climate or cyber, requires new approaches and additional calculations to try and quantify their impacts on the insurance portfolios. All these aspects make it increasingly important for insurance companies to have efficient models that can be run quickly, reliably and without incurring exorbitant costs.

MODEL DIMENSIONALITY AND PARALLELISATION

An ALM model does not usually contain very complex mathematical calculations. Nor does it require sophisticated algorithms. On the contrary, the atomic calculations are in fact rather simple, as an overwhelming majority of them are the four basic mathematical operations, spiced with an occasional exponential or logarithm. In spite of that, it can be a very costly computational problem due to its high dimensionality and interdependencies.

The high dimensionality comes from combining all the aforementioned elements of the projection, as shown in Figure 5.

FIGURE 5: CALCULATION FLOW IN DYNAMIC ALM MODEL

DIMENSION	TYPICAL RANGE
Number of model points	**10,000-500,000**
Projection length	**30-100 years (monthly)**
Number of stochastic simulations	**1,000-5,000**
Number of variables calculated	**200-400**

The number of model points might vary substantially among companies and models used, depending on several factors. For models using a dynamic ALM approach, there is usually a rather strict constraint coming from the performance or memory limitations—those models rarely use more than 25,000 model points in the total portfolio. For the liability projections part of the flexing models, some companies still chose to run them on the individual policy level. That is where the number of model points can go into millions. Afterwards, for the ALM part of the flexing models, liability cash flows are usually aggregated to a couple hundred to a few thousand liability segments.

On top of these considerations, asset valuations using a discounted cash flows method and any types of liability products, including annuities, disability or claim states, introduce a secondary time dimension.

While any of these dimensions individually are not particularly big, compared with data amounts processed by other industries or data science problems, the volume of the hypercube across all the dimensions combined can easily exceed the capacity of a lot of machines, even with powerful hardware by today's standards.

Many of the computational problems today are efficiently solved by parallelisation of calculations. Most computers nowadays have multiple cores that can be used to split up calculations among them. This is taken to the next level using the generalised computational capabilities of graphical processing unit (GPU) cards. In order to use parallelisation, however, the problem at hand must comprise many independent unit calculations and a limited number of dependencies or aggregations.

The ALM model problem, however, truly meets that requirement only across the stochastic simulation dimension and, in the case of a flexing model, also across model points. There are obvious dependencies among calculated variables and among the different time steps at which these variables are calculated. Therefore, historically ALM models were typically parallelised by stochastic simulations. It is also straightforward to achieve from a

model architecture perspective, using multiple central processing unit (CPU) cores or cores across multiple workers. As we will show in the following section, the higher level of available parallelisation with the GPU means that splitting only by stochastic simulations might not utilise the full potential of the available hardware. Also, GPUs excel in computing high volumes of simple vectorised uniform operations,[2] while there is a broader spectrum of various calculations across a single simulation in an ALM model.

In the case of a flexing model, for the liability side each model point can be independently projected across the whole projection period and only then can the aggregation happen—once for all periods. This provides an additional dimension of parallelisation that fits the GPU model better. Afterwards, the asset side can be parallelised across the stochastic simulation dimension, as the liabilities are on a more aggregate level at this step. The parallelisation benefit of this type of model will be greater.

In fact, a dynamic ALM model can still take advantage of parallelisation, but it will require a smarter design of the calculation flow and more sophisticated architecture. Even then, the performance benefits of parallelisation with GPUs will typically be smaller than in case of a flexing model and that will come at the cost of a more complex model. The more interruptions in the parallelised process, such as aggregations of data from different cores, the lower the parallelisation benefit will be. This will be consequential also for larger-scale models, in which a single stochastic simulation does not fit on a single machine due to memory requirements. In such cases, an aggregation across not only cores but also machines must take place at each calculation step.

IT landscape

The technical infrastructure is at the core of the modelling solution and as such it should be chosen carefully. Different architectures are available, and so the choice should be adapted to the company's and actuarial department's needs: on-premises, cloud solutions, CPU versus GPU etc.

ON-PREMISES SOLUTIONS

Customisation of the hardware platform can be of importance for maximising the performance of the modelling solution. As such, on-premises solutions offer high levels of customisation for the physical infrastructure on which the calculation will run.

Some of the impactful aspects that should discussed with the IT team are, for example:

- **CPU:** How many cores per CPU? Do we want to run massive parallel calculation on each worker node? What clock frequency is acceptable, as a higher clock frequency means lower run-time?
- **RAM:** What amount of random access memory (RAM) is sufficient for our use case? In the case of ALM modelling, having more RAM enables us to use more model points and perform more precise calculations, but it can also mean that we can process more simulations on each single machine.
- **GPU:** Do we want some part of the calculation to run on GPU? What kind of GPU do we need: how much GPU memory (vRAM) and how many GPU cores do we want on each machine?

With those different aspects that should be set up, on-premises solutions require a significant degree of involvement and collaboration between the team in charge of the model development and the IT team. Those kind of architecture decisions are generally well suited for long-term use.

The main advantage of an on-premises solution is that there is no pay-per-use scheme and the machines can be used as much as required without extra fees. However, there is a fixed capacity that cannot be easily and quickly adjusted and typically on-premise machines are not fully utilized, which gave rise to the popularity of cloud solutions.

CLOUD SOLUTIONS

Cloud infrastructures (e.g., AWS, Azure, GCP) provide elastic computing resources on demand with pay-as-you-go pricing that offer solutions fully managed by third-party providers, relieving IT teams of the need to purchase, install, manage and upgrade technology on-site.

Those solutions can scale easily to meet temporary workload demands and as such they can offer a flexible solution that would only be costly during phases requiring intense computing (e.g., during annual or quarterly calculations).

Compared to on-premises solutions, cloud platforms allow us to easily test different architecture, hardware configurations and solutions (e.g., using a 2,000-cores cluster for stochastic ALM calculation, machines with different memory sizes, GPU setups etc.).

[2] Vectorised operations function natively and act on multidimensional objects, e.g., arrays. They interface the low-level implementations of the underlying loop operations.

However, the flexibility and great features offered by cloud providers can come with a cost. Some features can introduce a vendor-locking mechanism, which adds a cost when migrating to another cloud provider: for example, storing the data in a custom database solution specific to the cloud provider.

MASSIVE PARALLELISATION USING GPU

Since Moore's[3] law started to wind down, GPUs have become a key part of modern supercomputers and continue to drive improvements in computing speed.

FIGURE 6: CPU-GPU COMPARISON

CPU	GPU
Several cores	Many cores
Low latency	High throughput[4]
Good for serial processing	Good for parallel processing

Architecturally, CPUs are composed of a few cores with lots of cache memory (temporary storage from which the processor can retrieve data more easily than from the RAM), whereas GPUs are generally composed of thousands of cores that can handle thousands of threads simultaneously. Figure 7 gives an overview of the number of cores and frequency for some of the most common household graphics cards.

FIGURE 7: GPU CHARACTERISTICS

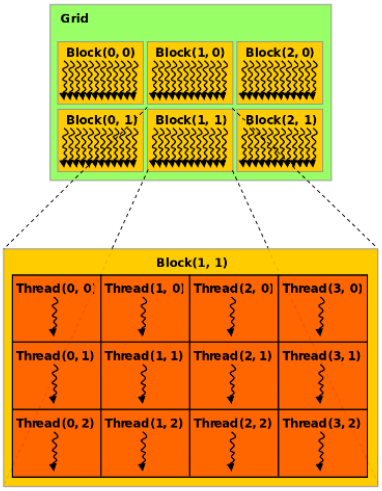
NVIDA CARD	NUMBER OF CORES	CLOCK SPEED	MEMORY
RTX-3050	2,560	1550 MHz	8 GB
RTX-3060 Ti	4,864	1410 MHz	8 GB
RTX-3070 Ti	6,144	1500 MHz	8 GB
RTX-3080 Ti	10,240	1370 MHz	12 GB
RTX-3090 Ti	10,572	1670 MHz	24 GB

Looking at these numbers, it is clear that the level of parallelisation enabled by the GPU is massive compared to the CPU, even on very expensive servers. However, to efficiently use this benefit of parallelisation, there must be enough independent calculation streams in the computational problem to solve. As shown before, this can be problematic for a dynamic ALM model.

Similarly to cluster or cloud computing on a CPU, multiple GPUs can also be combined into clusters—whether on a single machine or on multiple machines connected with each other. In either case, while parallel calculations inside the GPU are extremely fast, the transfers of data to and from the CPU to the device, or between devices, can be a bottleneck.

Due to the differences in architectures between CPUs and GPUs, programming on GPUs can seem more challenging at first. In the past, if users wanted to write arbitrary code that compiles and runs on a GPU, they would have had to use complex tool kits like CUDA or OpenGL, which provide additional tools on top of low-level programming languages such as C or C++. Those libraries are low-level solutions that allow for fine-grained control of the calculation. For example, the calculation can be specified at the thread block level: each thread block (or thread) can perform independent calculations in parallel with the other threads and, for each block, specific operators can be performed for efficient synchronisation between the different threads (e.g., for aggregating the results of the threads).

FIGURE 8: REPRESENTATION OF THE GRID, BLOCKS AND THREADS[5]



Thankfully, it is now possible to run code like traditional vectorised operations on a GPU without having to understand how programming using CUDA or OpenGL works. Some classical libraries available in Python have been adapted for GPU calculation (cf., Implementation in Python section below). However, in order to successfully choose parts of calculations to run on the GPUs and to develop truly optimised and performant code on the GPUs, one will still need to understand at least the fundamentals of GPU architecture.

[3] Moore's law states that the number of transistors that can be integrated into a circuit will double about every two years.

[4] CPUs are more efficient in switching between tasks done on different chunks of data. GPUs are built to operate on bigger chunks of data at once. Transferring data to the GPU memory has a higher overhead than in case of a CPU.

[5] Figure 8 is from the post "Thread Hierarchy in CUDA Programming," available at http://cuda-programming.blogspot.com/2012/12/thread-hierarchy-in-cuda-programming.html.

Implementation in Python

WHY PYTHON (AND WHY NOT)

Nowadays, most companies considering developing an in-house projection model often choose between C++, C# or Python as the language of choice. While C++ and C# have a huge native performance advantage, it is also much harder to find actuaries with a good knowledge of these languages. In recent years, Python has risen to be one of the most popular general programming languages, especially in data science and machine learning applications. It is widespread in overall science and technology university curricula, including actuarial programmes. It is a higher-level language than C++, C# or Java, with an easier syntax and language building blocks. Therefore, in this paper we focus on Python as a tool for building a projection model and consider several possible approaches to design and implement such a model in an efficient way.

Python is an interpreted language. As opposed to the compiled languages, it does not require compilation of the code prior to running it. Therefore, it provides the user with immediate evaluation. It is a huge advantage during code development. It allows the coder to enter an interactive Python session and run only the selected code snippets without the necessity of running the whole programme. Therefore, Python code is said to be developed in script files, which nicely reflects its interpreted nature.

Such on-the-fly code execution is extremely useful in shorter and ad hoc tasks like explorative data analysis. Indeed, an analyst can build the analysis step by step, by running and analysing smaller portions of the whole analysis process and unfolding the next analysis step based on the results of the previous steps. Another advantage of Python is that it relies on dynamic memory allocation. Because the code is translated into machine language during run time, the user does not have to predefine variable types and pre-allocate memory for them. It is a great simplification in code development, which is often a hassle in the compiled languages.

The interpreted nature of Python also brings several disadvantages. The most important ones are the speed and memory management. The execution time is a direct consequence of the fact that the code needs to be translated into machine language when it is being executed. In the compiled languages such translation happens prior to the execution.

Similarly, the Python interpreter does not know up-front about the nature of the variables in the code.[6] Therefore, all the memory allocations happen at run time, which is slower than pre-allocating memory at the beginning of the run based on the variables' known specifications contained in the compiled code.

Despite Python being highly suitable for ad hoc dynamic code development, it is frequently used in software development as a fully featured programming language. Indeed, the two disadvantages that we recalled are not impactful for many applications. However, if it comes to conducting a large number of operations and/or operating on large data sets they may be important factors.

In the actuarial projection models, we face both of these challenges. Indeed, the models can require calculating many characteristics at dense and long projections, and at the same time they are multiplied by the size of the portfolio, which can go into millions of inputs.

Raw Python data types, such as lists, tuples and dictionaries, can be cumbersome to work with for complex models. Indeed, on the one hand, the list objects are efficient but hard for development to store and operate on larger tabular data sets. On the other hand, user-friendly dictionary indexing is not efficient for extensive collections. Therefore, model development often relies on additional packages which introduce classes and functions that provide better objects' interface, programming logic or calculation efficiency. In the next section we describe several libraries which are frequently used for data processing and actuarial model development in Python.

For people familiar with Python and the libraries mentioned in the next sections from data science or machine learning context, it might be natural to try to approach a projection model as another data science problem, especially when using the libraries commonly associated with data science problems. But this is not a similar model and such a mistake can lead to an underestimation of the complexity of the dependencies and dimensions, as explained above in the Model Dimensionality and Parallelisation section of this paper.

USEFUL LIBRARIES

NumPy

NumPy is a very popular library for mathematical operations on non-scalar numerical data. It provides classes that implement vectors, matrices and multidimensional arrays. Furthermore, it contains efficient element-wise implementations of mathematical operations and reshaping as well as aggregation functions.

Contrary to standard Python, NumPy arrays are typed, and an array can contain only a single numerical data type. Methods operating on the arrays are implemented in C and based on highly efficient algebra libraries from Fortran. As a result, vectorised operations on NumPy arrays are almost as fast as code implemented directly in C/C++.

[6] Python even allows us to change the type of the variable completely unrestricted in the code.

Pandas

Pandas is the most used and most complete library for working with data frames. A data frame is a variable type which represents tabular data as commonly used in spreadsheets. Columns of a data frame correspond to distinct features or characteristics of a subject, or a phenomenon, and rows correspond to distinct subjects. Note that, depending on the convention, subjects can be observations of the same set of measurements in different time points.

As opposed to NumPy's array, Pandas's data frame can contain data of different types. Each column must contain one type of data, but different columns within the same data frame are not restricted to one data type. This way, we can collect both numerical and nonnumerical characteristics for each observation in one row.

Pandas implements many useful functions that can operate on entire columns similarly to NumPy. Furthermore, numerical columns are naturally integrable with NumPy operators. In fact, NumPy is a prerequisite for Pandas.

Pandas also offers flexible indexing of rows, including multi-indexing. At the same time, indexing can be utilised for efficient grouping operations. For instance, one data frame can contain observations for policies from different portfolios and by grouping one can easily calculate respective portfolio characteristics without splitting the data frame beforehand.

CuPy

CuPy is a drop-in replacement for NumPy, which enables calculations and storage of the arrays on a GPU. Using the same data structures, methods and syntax, it makes it extremely easy to bring NumPy code to the next level of performance using massive parallelisation provided by graphical cards. There is no required prerequisite knowledge of CUDA programming skills or understanding of the GPU architecture to be able to take advantage of CuPy. There are, however, some limitations of CuPy—not all NumPy functions might be available and other libraries building up on NumPy don't necessarily work directly with CuPy.

CuDF

Similarly to CuPy and NumPy, CuDF is a drop-in replacement for Pandas, which enables calculations and storage of the data frames on a GPU. No CUDA or GPU architecture knowledge is required, and many of the Pandas methods work directly with CuDF. There are some limitations, however, that might require modification of the original Pandas code in order to work with CuDF.

Numba

Numba provides an interface for low-level compiled code. Numba just-in-time (JiT) decorators allow us to indicate functions that need to be compiled the first time the function is called. Once compiled, these functions can run extremely quickly and efficiently. Therefore, Numba can be used to create efficient computing kernels that can be then called by other parts of the code.

Numba utilises the NumPy interface for the common object types like arrays and vectorised operations. Therefore, creating Numba code is often a straightforward copy from a corresponding NumPy code with appropriate decorators. Similarly, Numba offers effortless loop parallelisation. This way, with no additional code complexity, sequential loops can be split among the available computing CPU cores (on a local worker) to speed up computations.

Numba also provides a simplified way of writing CUDA kernels, which are functions compiled and executed on GPU cores. Similarly to normal CPU kernels, they support some basic Python and NumPy operations—and no CUDA C knowledge is required. There are more restrictions on the supported code, however, so usually the code has to be modified more to be compiled into a CUDA kernel than in case of a CPU Numba kernel. Compared to CuPy, Numba provides more low-level control of the CUDA kernels and the way they are executed on a GPU. A better understanding of GPU architecture and execution methods, however, is required to use the GPU efficiently.

Dask

Dask is a framework that enables splitting data frames over a cluster of multiple workers and parallelising operations on them, which is especially useful when the data frame size exceeds the memory capacity of a single worker. While most of the operations provided by Pandas are supported in the partitioned data frame approach, there are also some limitations, which might require modifications of the original Pandas code before it works with Dask.

Dask also provides an automated task scheduler, which can derive a task execution tree based on the delayed operations defined on the data frame(s). The scheduler will detect dependencies and operations that can be parallelised in order to process the data in the most efficient way on the predefined Dask cluster.

There is also a Dask-CuDF module, which brings Dask functionalities to clusters of GPUs based on the CuDF library.

Others

A plethora of other Python libraries provide functionalities similar to the ones mentioned above. While we focus on the libraries listed above, some of the following libraries could be used as their replacements, and perhaps in some cases could lead to even better results.

Ray is one of the alternatives to Dask for distributed computing over a cluster of machines. The way it works is different, and it does not provide the same features regarding task scheduler or built-in data frame support, but it allows a greater level of control over the way tasks are managed, ordered and distributed.

Modin is another framework that seamlessly allows Pandas-based code to run in a cluster environment using Dask or Ray as its engine.

Xarray is an extension of the NumPy array data structure, which allows user-friendly naming of the array dimensions and elements, instead of simple numerical indexing.

PyCuda is a lower-level CUDA interface for Python, which allows writing and executing CUDA kernels inside Python code. While it enables the most customisable GPU code execution, it also requires much more knowledge of GPU architecture and CUDA C to be used efficiently.

There are also several machine learning frameworks that enable GPU computations, like TensorFlow, Keras or PyTorch. As they are focussed on specific ML-related applications, they are less useful for developing projection models.

USING DATA FRAMES – PANDAS AND CUDF

In this study, the following two libraries were used to compare CPU and GPU performance of solutions with architecture based on data frames: Pandas for CPU and CuDF for GPU. Those two libraries implement the same application programming interface (API) based on Pandas. As such, we could write the same code and easily switch from one environment to the other (either GPU or CPU).

However, there are some differences between CuDF and Pandas, both in terms of API and behaviour, so we were restricted to a common basis between the two APIs. As an example, CuDF doesn't support iterations over a data frame because it would require a lot of data transfer between the GPU and CPU, which would yield extremely poor performance. More importantly, the row ordering is not guaranteed after *join, merge or groupby* operations with CuDF compared to Pandas and the multi-index querying is not fully functional.

We have developed an implementation where we vectorise as much as possible the calculation by using the classical Pandas operators, optimising for performance, but in the meantime we took into account the readability and maintainability of the code. To meet those criteria we have chosen to not include the time as a dimension in our data structures, so the data in our data frames is replaced at each time step with the new calculated data.

With this convention, the code was exactly the same between the CuDF and Pandas versions, with only the imports differing.

USING ARRAYS – NUMPY AND CUPY

As an alternative to the data frame-based approach, we have also investigated a solution using NumPy, which is the cornerstone of fast numerical calculations in Python.

We have used a code and data structure similar to those in the data frame-based approach. However, the limitation on dimensionality of the projection data structure is lifted when using NumPy arrays. It is possible to have an array that covers all the necessary dimensions—policies, variables, projection time steps and stochastic simulations. As this approach might cause issues with such an array exceeding the memory capacity of a single worker and our model implementation still focussed on the liability projection for flexing, we have chosen to leave the simulation dimension out of a single worker solution. In this kind of model, simulations are only relevant in the asset module performing flexing, where the liability policy dimension is already reduced. In a dynamic ALM model, simulations can be added at the next level of architecture—when dealing with a cluster of workers and splitting their workload by simulation.

We have developed a solution that can either keep the results of an arbitrary number of projection steps in memory, or overwrite them from period to period, only keeping the most recent time step. This was sufficient in the simple model, where in the calculations we didn't have cross-time references larger than one period.

The low-level precompiled code providing vectorised operations over arrays and broadcasting to match dimensions across which operations are performed means that calculations across model point dimensions can be done very efficiently. The projection time dimension was handled using a simple for loop, as this is a sequential calculation with clear dependency between each iteration. The order of model variables' calculations was set manually.

To make the code more readable, we've used Python dictionaries to map the dimension indices of different variables to user-friendly names. This had no significant impact on performance.

The GPU-enabled equivalent of this model using CuPy required very limited adjustments to the code—99% of the code base remained unchanged after just changing the references to the NumPy library to those for the CuPy library. The NumPy arrays were also replaced by CuPy arrays, so all the data was directly stored in the GPU memory and didn't require further transfers between the host and device memory.

PRECOMPILED SOLUTIONS USING NUMBA

As a third option, we have explored the pre-compiled code solution using Numba. In principle Numba is also based on NumPy data structures and works well with NumPy arrays.

There are two types of Numba decorators that slightly modify the behaviour of the underlying functions. The first type is the *@JiT* decorator, where the whole argument array is passed to the underlying function and it is up to that function to loop across all elements of the array and apply the calculations appropriately. The second type is the *@vectorize* decorator, where Numba applies a NumPy-style vectorisation and automatically calls the underlying function many times on each element across a selected dimension. In our implementations, we have focussed on the JiT decorator approach. This meant adding an explicit second for loop across the model point dimension in the period projection calculation function. In vanilla Python using loops is generally inefficient but, in this case, thanks to the code being compiled into low-level machine code, loops become almost as efficient as in C or C++.

There are, however, some limitations as to what can be converted into the low-level code by Numba, and even though the version of the library used claimed to have support for typed dictionaries, using them in the Numba kernels leads to a massive performance deterioration. Therefore, we were forced to remove the dictionary-based mappings for variable indices and refer to them using raw numerical indices. This improved performance by an order of magnitude but came at a cost of worse code readability.

By making a very simple adjustment to the for loop across policies, it is possible to enable the multi-core processing capability of Numba, which can take advantage of all the cores of the local worker.

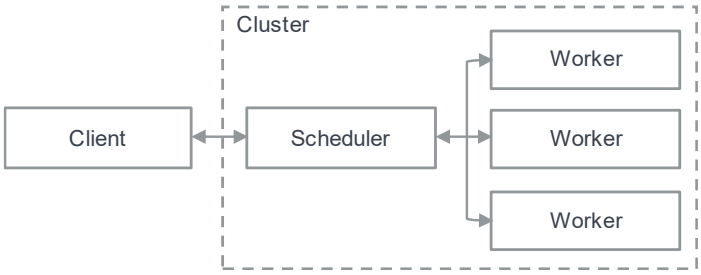
The CUDA JiT version of the model was largely the same as the CPU JiT version. The main difference was that it required specification for the GPU regarding the shapes of the grids and blocks of threads. The inner loop across policies could also be removed, as the way the GPU works with Numba kernels using the CUDA JiT decorator is that each element across the dimension being parallelised gets its own GPU thread to calculate it. That means that all policies are split up into chunks equal to the number of threads multiplied by the number of blocks and each chunk is calculated fully in parallel, while different chunks are sent sequentially to the GPU. This is very efficient, as long as there are enough policies that can provide sufficient occupancy of the GPU threads. All arrays used in the calculations were explicitly transferred to the device GPU memory before calling CUDA kernels, to have full control of the memory data transfers.

We have not done any optimisations of the CUDA JiT implementation. It is possible to get better efficiency by utilising different types of GPU memory that are available and managing the size of blocks. In both Numba approaches, further optimisation could also be done by omitting calculations for policies that have already reached their maturity terms. This is not easily done when performing vectorised calculations on the whole portfolio at once—as in case of data frame-based and Numpy array-based solutions.

MANAGING CLUSTERS AND OUT-OF-MEMORY DATA

As an alternative when the data is too large to fit in memory, we have investigated the use of Dask, which implements Pandas-like features that allow us to distribute the calculation over a local or remote cluster. Dask is not only limited to a data-frame API, but also implements a NumPy-like API and the possibility of executing remotely any Python function on a cluster.

FIGURE 9: DASK CLUSTER



The cluster is composed of a *scheduler*, which is responsible for distributing the calculation tasks among the different available *workers*. Those workers can have an arbitrary number of CPU cores and GPUs depending on the hardware used and how Dask is set up.

We used the data-frame API from Dask to execute the calculation even when the data cannot fit in memory. By using the Dask API we can specify how to partition the data at read time and for the rest of the calculation, so that each partition is stored in memory only when needed. Dask also has default parameters that are used to automatically define partitions and avoid running out of memory. Before running the calculation, Dask also builds and tries to optimise the calculation graph that will be processed by the scheduler. This can be useful to investigate potential bottlenecks or issues in performance.

The benefits from using Dask also come with a cost as some features specific to the Pandas API are not available in Dask. As an example, Dask doesn't support multi-indexing or value assignment to specific rows. Thus, similarly to the CuDF approach, we decided to not include the time dimension in our data structures. The data frames are updated at each time step with the new calculated data.

Benchmarks

The model used in this benchmark is a simplified liability model for savings products with guarantees, common in the European market, which projects the following variables at the model point level with a monthly projection step:

- Mathematical reserves, which correspond to an accumulated savings fund calculated retrospectively
- Cash flows:
  - Premium income
  - Death, lapse, partial lapse, maturity outgoes
  - Investment income and gross profits
- The expected evolution of the policy takes into account:
  - Death rates with mortality tables by age
  - Lapse and partial lapse, which are defined by policy year
- Cost of guaranteed interest

Expenses and commissions have been set to zero in the simplified model. The model is equivalent to the liability component for a flexing ALM model. We have run the model with different numbers of policies with varying sets of assumptions.

As mentioned in the introductory sections above, we believe this projection model is a fundamental block required in all ALM models. Building upon this, depending on the preferences and functional requirements of the ALM block, the models can go different directions and therefore utilise different architectures. The benchmark can be extended to cover those different options, but we believe that in most cases the choices with respect to these options will have limited impact on the relative final performance. We think that the results obtained with the chosen simple model are generally representative for more complex models too, as far as orders of magnitude are concerned. However, in the dynamic ALM model case, the increased architectural complexity will likely lead to lower benefits achieved thanks to massive parallelisation with the GPU computing. In that case, more expertise is required in the design phase to come up with an architecture that will maximise the possible gains, taking into account all the specific requirements.

The machine used in our benchmarks had the following specifications:

- Intel Core i7-12700K (16 cores) with 32 GB RAM
- Nvidia RTX A4000 GPU (6,144 CUDA cores with 16 GB RAM)

Code workflow

The data model and code workflow were created with two main goals in mind. The first was performance, by vectorising as much calculation as possible, and the second was to have a code that is easy to read and maintain.

For each time step, where applicabl e, the different calculations are vectorised so that we can take advantage of faster calculation with pre-compiled library code.

FIGURE 10: CODE WORKFLOW

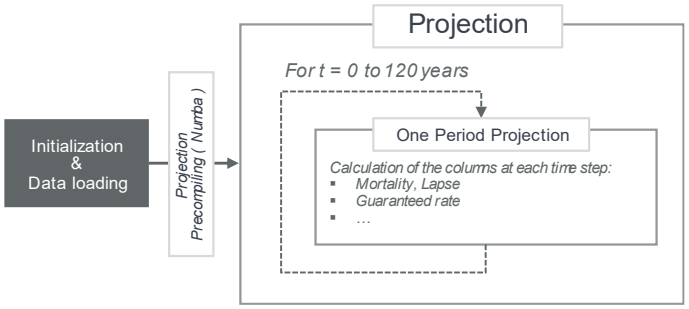
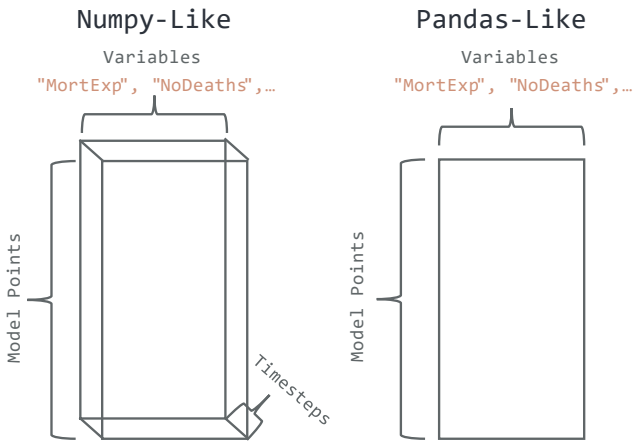


FIGURE 11: DATA STRUCTURE (NUMPY VS. PANDAS)



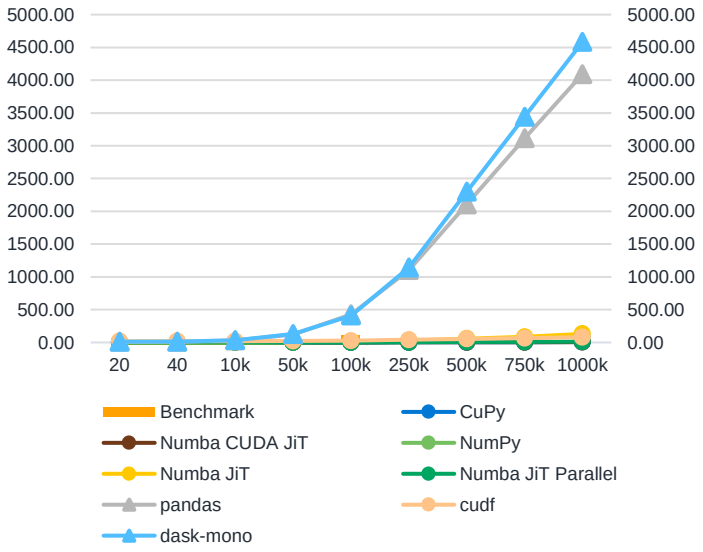
For our benchmark comparison we implemented the model with a mono-core CPU (NumPy, Pandas, Numba JiT), multi-core CPU (Numba JiT parallel) and GPU-based libraries (CuPy, CuDF, Numba CUDA).

We have also compared the performance with a reference benchmark using a representative experience of a third-party projection model software. It should be noted that this is a synthetic benchmark and various individual proprietary platforms will likely have different results.

In all the Python benchmark results, we have focussed on the pure calculation run time—excluding elements such as reading in data, compilation of the code, allocation of memory for the data structures used or writing out results. The time for reading in data and allocating memory for the projection results is, of course, proportional to the size of the data, but for a given size it is fairly comparable across all methods. As an example, those two steps for 100,000 model points (MP) data take jointly around one second. What is worth noting, when using GPU computations with more complex model architecture, is that, if the amount of transfers between the CPU memory and GPU memory increases relative to the number of computations performed by the GPU on that data, then it could affect the run time adversely.

Our results are summarised in Figures 12 to 15.

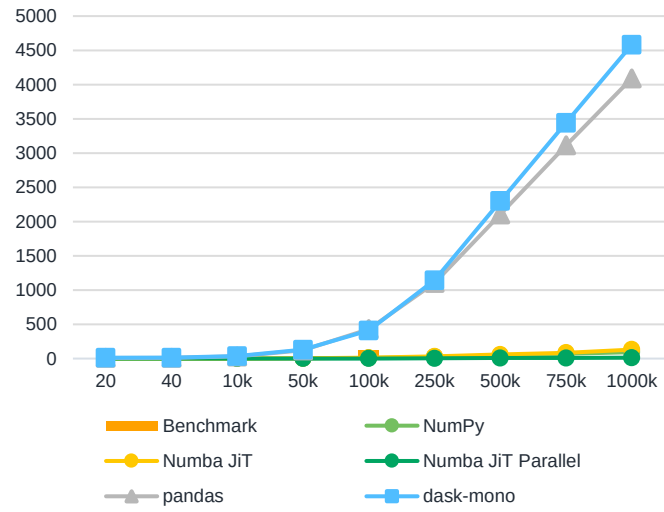
FIGURE 12 COMPARISON ACROSS ALL IMPLEMENTATIONS



Note: Elapsed time in seconds as a function of the number of model points.

As we could expect, the run time scales linearly with the number of model points for CPU (single core) implementations. However, we observe a significant difference in how the run time scales with the number of model points between GPU and CPU implementations, especially on Pandas-like implementations (Dask and Pandas): the run time increases significantly faster on Pandas-like implementations. For 100,000 model points, Pandas implementation takes significantly more time than the reference benchmark (see Figures 12 and 13).

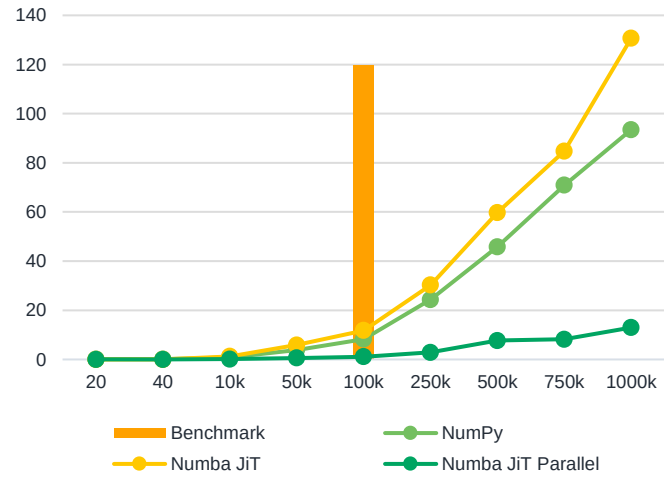
FIGURE 13 COMPARISON ACROSS CPU ONLY IMPLEMENTATIONS



Note: Elapsed time in seconds as a function of the number of model points.

When excluding Pandas-like implementations we observe that the NumPy and Numba implementations are significantly faster than the reference benchmark for 100,000 model points (see Figure 14).

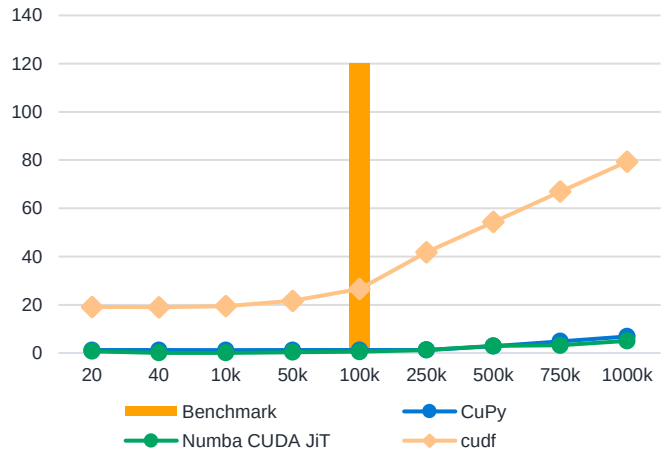
FIGURE 14 COMPARISON ACROSS CPU (NUMPY-LIKE) IMPLEMENTATIONS



Note: Elapsed time in seconds as a function of the number of model points.

The implementations using a GPU with a single-core CPU show again a significant difference in how the run time scales between the Pandas-like library (CuDF) and the NumPy-like libraries. (see Figure 15).

FIGURE 15 COMPARISON ACROSS GPU WITH SINGLE-CORE CPU IMPLEMENTATIONS AND BENCHMARK (SINGLE-CORE CPU)



Note: Elapsed time in seconds as a function of the number of model points.

FIGURE 16 COMPARISON ACROSS GPU WITH SINGLE-CORE CPU IMPLEMENTATIONS AND BENCHMARK (SINGLE-CORE CPU)

NUMBER OF MODEL POINTS	40	10K	50K	100K	250K	500K
Benchmark	-	-	-	120	-	-
Dask mono-core	13	35	129	407	1143	2298
Pandas	11	31	120	430	1104	2105
CuDF	19	20	22	27	42	54
Numba JiT	0.01	1.2	5.9	12	30	60
NumPy	0.12	0.87	3.86	8.26	24.4	45.8
CuPy	1.2	1.18	1.21	1.22	1.23	2.82
Numba CUDA JiT	0.03	0.08	0.31	0.55	1.28	2.93
Numba JiT Parallel	0.01	0.12	0.57	1.13	2.87	7.68

FIGURE 17 RUN SPEED FOR 100,000 MP RELATIVE TO THE BENCHMARK (HIGHER = FASTER)

MODEL	ARCHITECTURE	SPEED FACTOR
Benchmark	Single-core CPU	1.00
Dask mono-core	Single-core CPU	0.29
Pandas	Single-core CPU	0.28
CuDF	GPU (6144 CUDA cores)	4.51
Numba JiT	Single-core CPU	10.17
NumPy	Single-core CPU	14.53
CuPy	GPU (6144 CUDA cores)	98.36
Numba JiT Parallel	Multi-core CPU (16 cores)	106.19
Numba CUDA JiT	GPU (6144 CUDA cores)	218.18

Conclusions

Designing and implementing a projection and ALM model in-house from scratch is a complex task that involves many potential traps for the unaware. Making it a production-grade industrialised tool, integrated efficiently with the other processes in the reporting flow, adds even more challenges. With a team comprising experts both in actuarial modelling and IT, good preparation and awareness of those challenges, that goal is achievable. In that case, the results might be every bit as good as expected. As our benchmarks show, using Python CPU calculations it's possible to achieve model performance improvements of factor 10, and with Python GPU calculations it's possible to achieve performance improvements of factor 200, compared to the selected benchmark experience.

The functional requirements and closely related choice of architecture, data structures and libraries involved largely determine the limits of performance of such models. Provided resources with the necessary expertise and making well thought through choices, it is possible to build a high-performance ALM model even with Python, which is also not very complex from the technical and code base perspectives.

As shown by our benchmarks, if performance is the key priority, then utilising GPUs is by far the most efficient solution for large portfolios, when a large parallelisation benefit potential is present. Its strength will decrease if the parallelisation potential will not be on par with the GPU capacity for parallel computations. This can be due to many reasons, such as complex architecture (dynamic ALM models), smaller portfolios or a low number of stochastic simulations. Therefore, a careful design phase is key to take the maximum possible advantage of the available technology and match the dimensionality of the computational problem with the available hardware parallelisation potential. From tested Python libraries, Numba is the winner in terms of pure performance. However, it comes at a cost of a more complex code base to manage. When simplicity of the code is equally important, solutions using CuPy might be a better option, as in our tests they sacrifice little performance (and virtually none for larger data sets), but provide a simpler, intuitive coding interface familiar to anyone who has used NumPy.

If speed is important, but simpler architecture is desired, if required architecture limits the benefits of the GPU parallelisation or if the data dimensions are not large enough, e.g., thanks to clustering, then CPU architecture might be an acceptable solution that can still outperform dedicated third-party modelling platforms.

There are many improvements and considerations that could further enhance this analysis—such as further code optimisation, adding another aspect of architecture and the potential for the parallelisation dimension to see multi-user usage of the models or sets of runs that usually have to be calculated jointly at any given reporting period. Additional interesting items would be benchmarking efficiency of memory usage or input-output operations, such as reading in data and writing out results, using different architectures and libraries, automation and integration with other steps of the run process and other model-related processes in general, as well as designing and developing a dashboard that would simplify the setup and use of the models. Most of these aspects are also easier and more flexible to do when having a custom in-house model.

It is likely that, over time, the libraries mentioned in this paper will evolve further and include additional functionalities, or that new, better libraries will appear.

Therefore, the most important conclusion should be that there is no universally best solution for architecture and implementation of an ALM model and each case should be thoroughly analysed with the support of experts in actuarial modelling, architecture and efficient IT implementations. A clear cost/benefit analysis should be performed, and different proprietary solutions should certainly also be considered, as building a custom in-house model will not be the best solution in every case. If you are considering modernising and industrialising your modelling solution, whether upgrading a proprietary platform or building an in-house model, you can contact the authors or your local Milliman consultants for helpful expertise and a more specific discussion on what to consider and what might work best in your particular case.

Milliman

CONTACT

Karol Maciejewski
karol.maciejewski@milliman.com

Mehdi Echchelh
mehdi.echchelh@milliman.com

Dominik Sznajder
dominik.sznajder@milliman.com