MILLIMAN WHITE PAPER

# GPU: Exploring use cases in actuarial modeling

Introduction, applicability, benchmarks, and cost-performance overview.

Karol Maciejewski
Chad Schuster
Jim Brackett
Corey Grigg

Milliman

Graphical processing units (GPUs) have reached new heights of popularity with the advent of artificial intelligence (AI) models in recent years, which would not have been possible without them. They provide unmatched computational power for parallelizable calculations, hundreds of times faster than traditional CPU-based computing. Can they be effectively used for actuarial modeling, and if so, for which use cases and how?

## A brief history of a GPU

Modern GPUs appeared at the end of the 1990s, when 3dfx and NVIDIA released their first graphics cards with a dedicated processor to perform graphics-related calculations. This was a revolutionary idea: offloading some of the heavy calculations from the computer's central processing unit (CPU) and freeing it for other tasks, at the same time allowing much faster graphics calculations due to the new processor's specialization. The initial uses of hardware-accelerated graphics cards were video processing and gaming (particularly 3D games).

During the 2000s, the spectrum of potential applications for GPUs has dramatically increased with the introduction of CUDA and OpenCL, two frameworks that allow users to submit custom calculations to be computed on the GPU, not restricting users to the predefined graphics routines implemented by the chip producers. CUDA (originally: Compute Unified Device Architecture, although nowadays, it is rarely expanded) is a proprietary standard created by NVIDIA and is supported by all its devices, providing implementations in several popular languages, such as C, C++, C#, Fortran, Python, or Julia. OpenCL is an open standard supported by many companies (including NVIDIA and AMD). Currently, with NVIDIA achieving an almost monopolistic position in the GPU market, CUDA has emerged as a more popular choice for GPU computing applications. The introduction and evolution of these standards have paved the way for broadening the popularity of general-purpose GPU computations (GPGPU). In 2016, AMD released ROCm, an open-source equivalent of CUDA for AMD GPUs, and although gaining traction, it is still significantly less popular than CUDA.

Over the years, as implementations and libraries in more languages were added, CUDA gained increasing popularity in computational physics, biology, chemistry, and finance. Its first significant expansion to a wider audience came in the second half of the 2010s, together with blockchain and cryptocurrency hype, in which GPUs were heavily used for cryptographic calculations. An even greater explosion in GPU use came in the 2020s with mass adoption of a new wave of GPT AI models (generative pre-trained transformers), which currently require GPUs for training and efficient execution of user queries, exploiting computations orders of magnitude faster than those achievable on CPUs.

## CPU versus GPU

So, what makes a GPU so much faster than a CPU? A key aspect is a fundamentally different architecture. A typical CPU consists of a few (up to several dozen, in the case of advanced server processors) calculation cores, each extremely fast and able to compute complex tasks sequentially. The GPU, on the other hand, comprises thousands of cores, each of which can accommodate a subset of specific "simple" calculations; this arrangement opens the door to massively parallel execution.

One can think about the following analogy, where the CPU is a sports car and the GPU is a cruise ship. Obviously, in a typical situation, getting from point A to B is faster with a sports car. However, if one needs to transport (tens of) thousands of people, being able to do so with one trip of the cruise ship significantly outperforms making thousands of trips with the sports car (if, of course, we assume that both points A and B can be reached by the car and the ship).

The above thought exercise also highlights another important characteristic of a GPU—all the passengers are traveling on the same route. Translating this back to computational terms, a GPU can execute highly parallel calculations by assigning each of its many cores to a data element, provided that the same operation is performed on each of them. This model of calculations is called SIMD (single instruction, multiple data).[1]

Historically, PC CPUs have been single-core and focused on sequential execution. Over the years, they have been extended to have multiple cores and, in more recent years, even to include some SIMD-like instructions (for example, several versions of the AVX instruction set support vectorized operations). The base architecture, however, is subject to limitations on how dramatically it can change, as it needs to be able to handle legacy cases. GPUs, on the other hand, have been designed from scratch with a different purpose and execution model in mind, so they are still in a different league with respect to parallelizing calculations.

Nowadays, there are many manufacturers (even if one is in a clearly dominant position), models, and generations of GPU cards available. Are all GPUs equal? Absolutely not. So, what is the best GPU to run a particular set of computations? The answer depends on the type of calculations needed and requires a deeper understanding of the GPU architecture and the different types of calculations it can handle.

## Understanding GPU performance

The first point is the general-purpose calculation units themselves. There are several types of contemporary GPU cards. The main ones, and historically the first, are called CUDA cores in NVIDIA cards and Stream Processors in AMD cards. Although the naming differs between manufacturers, the concepts and functionalities are largely the same. In the remainder of this paper, we will use NVIDIA naming for conciseness. Typically, the number of CUDA cores constrains the maximum level of parallelization that can be achieved; more cores mean higher performance. This is partially true. In fact, there are several types of CUDA cores in each GPU, specializing in performing calculations on different types of numeric data. The number of CUDA cores reported as a single number in most marketing materials and simple technical specifications refers to the cores specialized for single-precision floating-point calculations (denoted FP32, with 32 bits used to represent the number in computer memory). There are, however, several other types of numbers, each with a different type of CUDA core—for example, INT32 for integer numbers, FP16 for half-precision floating point numbers, or FP64 for double-precision floating point numbers. In most current consumer- and even workstation-level GPUs, the number of FP64 cores is lower than the number of FP32 and lower-precision cores. This disparity was present even in first-generation GPUs but has been amplified in more recent generations, as the most common uses of GPGPU were focused on computations that do not require such high precision (e.g., most machine learning applications and generative AI use). Boosting performance in these areas (by increasing the number of relevant cores) has necessitated a sacrifice in other areas. Unfortunately, scientific (and actuarial) calculations typically require higher precision, so FP64 performance is much more important. Although top-of-the-line server-grade GPUs in each generation have retained this capability for high-precision numbers, each successive generation comes with a heftier price tag.

Given this trend, it turns out that a server-grade GPU from several generations ago might outperform some of the much more expensive current consumer or professional GPUs in high-precision calculations. To compare different devices' compute capabilities, instead of looking at the number of cores, it can be more instructive to look at FLOPS (Floating Point Operations Per Second), which, for GPUs, are typically measured with giga- or tera- prefixes ($10^9$ and $10^{12}$, respectively).

**FIGURE 1: SELECTED GPU COMPUTE CAPABILITIES OVER TIME**

| GPU | YEAR | TIER | F32 TFLOPS | F64 TFLOPS |
|---|---|---|---|---|
| NVIDIA 8800GS | 2008 | Consumer | 0.26 | N/A |
| ATI Radeon HD 4870 | 2008 | Consumer | 1.2 | 0.24 |
| NVIDIA GTX 980 | 2014 | Consumer | 4.98 | 0.16 |
| AMD Radeon R9 295X2 | 2014 | Consumer | 5.73 | 0.72 |
| NVIDIA Tesla K80 | 2014 | Server | 4.11 | 1.37 |
| AMD FirePro W9100 | 2014 | Server | 5.24 | 2.62 |
| NVIDIA Tesla P100 | 2016 | Server | 9.53 | 4.76 |
| NVIDIA RTX 2080 | 2018 | Consumer | 10.07 | 0.31 |
| NVIDIA Tesla V100 | 2018 | Server | 14.13 | 7.07 |
| NVIDIA A100 | 2021 | Server | 19.49 | 9.75 |
| NVIDIA RTX 4090 | 2022 | Consumer | 82.58 | 1.29 |
| NVIDIA H100 | 2022 | Server | 51.22 | 25.61 |
| AMD Radeon MI300X | 2023 | Server | 81.72 | 81.72 |
| NVIDIA H200 | 2024 | Server | 60.32 | 30.16 |
| NVIDIA RTX 5090 | 2025 | Consumer | 104.8 | 1.64 |

Source: Compiled based on TechPowerUp GPU database[2]

1. In fact, the model utilized by GPUs is its close relative, called single instruction, multiple threads (SIMT). Explaining the technical differences between these are outside the scope of this paper.

2. TechPowerUp. (n.d.). GPU Specs Database. Retrieved September 24, 2025, from https://www.techpowerup.com/gpu-specs/.

In a sample from hundreds of models over the years, it is clear that contemporary consumer GPUs provide the best value for F32 calculations, whereas 10-year-old server GPUs still outperform them for F64 calculations. In the last few generations, an even stronger emphasis has been placed on the F16 and on new types of cores, namely tensor cores (NVIDIA) or matrix cores (AMD); these cores are heavily utilized in generative AI applications (and for brevity are not shown in the table above).

Tensor cores perform, in a single processor cycle, a fused multiply-add (FMA) operation on matrices, bringing parallelization to another level. Not every computational problem relies on or can be reformulated to this kind of operation, but the recent generative AI models depend heavily on it. As generative AI is responsible for most recent GPU demand, it is no surprise that manufacturers have been focusing on this new type of core. However, unless an actuarial model can be formulated as a series of matrix multiplication and additions, this feature of the latest GPUs will be of limited usefulness in the actuarial domain. Even for cases where a model can leverage FMA operations, tensor cores are specialized for specific types of numbers, typically accommodating lower-precision cases suitable for AI. Accordingly, high-precision FMA operations required for actuarial modeling will still be significantly less performant.

Another consideration is memory availability in the GPU device (VRAM, from the original Video RAM). By design, GPUs require data to be in dedicated device memory and not in the general system memory (RAM). This means that the more VRAM the device has, the more data it can store and the more calculations it can execute in parallel (this is true for data-intensive problems, which have relatively large data input per computation unit). There are also differences between GPU generations and models regarding the type of memory used and the corresponding data throughput, and server-grade GPUs tend to have more advanced, faster memory dies. As we will see later in this paper, this can also make a substantial difference for some applications.

The computational power of GPUs comes with an increasingly high price tag. Before the cryptocurrency boom, a top-tier consumer GPU would cost significantly less than $1,000 (e.g., NVIDIA GTX 1080 released in 2016 cost $599 at launch[3]), whereas a server-grade GPU would be around $5,000 (e.g., NVIDIA Tesla P100 released in 2016 cost $5,699 at launch[4]). However, since that time, there have been three notable events contributing to an extreme rise in price:

- Demand surge due to cryptocurrency boom
- Chip shortage and logistic problems during COVID pandemic
- An even bigger demand increase due to the dawn of generative AI

As a result, the recent top-tier consumer card, such as the NVIDIA RTX 5090, launched at $1,999, and a server-grade GPU, such as the NVIDIA H200 or AMD MI300X, will cost over $30,000. New GPU cards also have higher power consumption requirements, which additionally makes them more expensive to run and cool. For a company that utilizes cloud resources, it is certainly the case that the hourly cost of a GPU-equipped resource will be higher than that of a CPU-only calculation resource. The ratio of GPU-to-CPU hourly cost will vary based on the resources that one is utilizing, but whatever the selection, a comparison of total cost is clearly an important consideration.

So, is it still worthwhile to go down the GPU path? What kind of real-world performance improvements are possible? There are many factors influencing this, and there are several things to keep in mind when faced with claims of "N times performance gains." In the table below, we highlight a few key aspects:

**FIGURE 2: GPU-CPU PERFORMANCE GAIN FACTORS**

1. **Computational problem**
   The type of computational problem at hand will determine the possible performance gains. Some problems are much more suitable for GPU computing than others, as we will discuss in the next section of the paper.
2. **Reference CPU count**
   What is the base CPU core count for the benchmark? The same use case might show a different gain factor if compared to a single CPU core or multicore run (typical for compute-heavy problems).
3. **Quality of the reference CPU code**
   Level of optimization of the original CPU code. Often, the GPU code is written by more performance-aware developers and is simply better than legacy CPU code. In some cases, significant performance gains could be achieved just by refactoring the original CPU code.
4. **GPU used**
   What generation and type of GPU is used in the benchmark? As shown in Figure 1, there are significant performance differences between GPU devices, so a benchmark from 2012 would likely underestimate today's potential gains.
5. **Sample bias**
   Use cases with higher performance gains are more likely to be published or publicized.

3. TechPowerUp. (n.d.). NVIDIA GeForce GTX 1080. Retrieved September 24, 2025, from https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839.

4. TechPowerUp. (n.d.). NVIDIA Tesla P100 PCIe 16 GB. Retrieved September 24, 2025, from https://www.techpowerup.com/gpu-specs/tesla-p100-pcie-16-gb.c2888.

Bearing in mind the above, it is true that GPUs can provide a hundredfold increase in performance. In many cases, the gain factors will be lower, however, either because of the nature of the problem itself or because of the implementation on the GPU. As with CPUs, there can be a substantial difference between "GPU code" and "efficient GPU code," and the latter requires a deep understanding of the GPU architecture. We will touch upon this topic in the last section of this paper.

In Figure 3, we present selected examples of speed-ups across different application areas that have been described online.

**FIGURE 3: EXAMPLE SPEED-UPS IN REAL-WORLD APPLICATIONS**

| USE CASE | CPU CORES | GPU USED | SPEED-UP |
|---|---|---|---|
| **Protein structure inference[5]** | 128 | NVIDIA L40S | 177x |
| **Graph community detection[6]** | 112 | NVIDIA H100 | 47x |
| **Simulating stock market mid-price for HFT[7]** | N/A | NVIDIA H200 | 114x |
| **Weather forecasting (WRF-WSM5)[8]** | 1 | NVIDIA C2050 | 403x |

Data sources: See footnotes.

There is also an interesting presentation available on the NVIDIA website[9] highlighting performance gains across many use cases in various domains (biology, chemistry, physics) from using GPUs. However, as it was last updated at the end of 2012, the numbers would likely be very different when reevaluated on today's GPUs.

# Suitability for GPU processing

We have already indicated in the earlier sections of this paper that some computational problems are more suitable for GPU processing than others. Although the "best" group might exhibit performance improvements of hundreds of times, the "worst" will very likely show performance deterioration.

Three main elements are key in determining if a given problem is worth the while implementing on a GPU:

**FIGURE 4: KEY ASPECTS OF GPU SUITABILITY**

1. **Data size**
   There needs to be **sufficient** data to saturate thousands of GPU cores and offset the cost of transferring it to the device VRAM. At the same time, the problem ideally should be more compute-intensive than data-intensive.
2. **Independence of calculations**
   Ideally, the calculation for each data element should be independent of the other elements.
3. **Degree of code branching**
   The code executed in parallel should be as uniform across GPU cores as possible. Many conditional paths that depend on data might decrease performance.

The first aspect relates to the fact that there is a (quasi-)fixed cost of initializing calculations on the GPU. If the problem is too small, the initialization itself will take longer than solving it on a CPU, so the performance will be, in fact, worse. A dominant component of that initial cost can be the transfer of data to the dedicated memory of the GPU. Even though RAM is considered to be the fastest possible data storage option, transferring data between the host (general computer RAM) and the device (GPU RAM) introduces latency that becomes significant if subsequent compute times are too short.

The second aspect is directly related to the GPU architecture. As mentioned, the GPU consists of thousands of computational cores, and its power therefore lies in its being able to process calculations in a massively parallel way. If calculations on different cores must stop and wait for other cores to exchange information, or if not enough cores are able to process the calculations simultaneously, this will leave the GPU underutilized and therefore not efficiently used. The best problems for the GPU will consist of many independent atomic calculations, each of which can be completed by a single GPU core—such as independent Monte-Carlo simulations, modeling individual policies, or performing the same element-wise calculation on large arrays of data.

5. Technical Blog. (2025, September 10). Accelerate protein structure inference over 100x with NVIDIA RTX PRO 6000 Blackwell server edition. NVIDIA Developer. https://developer.nvidia.com/blog/accelerate-protein-structure-inference-over-100x-with-nvidia-rtx-pro-6000-blackwell-server-edition/.
6. Technical Blog. (2025, September 23). How to accelerate community detection in Python using GPU-powered Leiden. NVIDIA Developer. https://developer.nvidia.com/blog/how-to-accelerate-community-detection-in-python-using-gpu-powered-leiden/.
7. Technical Blog. (2025, March 4). GPU-accelerate algorithmic trading simulations by over 100x with Numba. NVIDIA Developer. https://resources.nvidia.com/en-us-financial-service/en-us-financial-services-industry/gpu-accelerate-algorithmic.
8. Ridwan, R., Kistijantoro, A. I., Kudsy, M., & Gunawan, D. (2015). Performance evaluation of hybrid parallel computing for WRF model with CUDA and OpenMP. IEEE Xplore. https://ieeexplore.ieee.org/document/7231463.
9. Berger, M. (2012, December 21). NVIDIA computational chemistry & biology. NVIDIA. https://www.nvidia.com/docs/IO/122634/computational-chemistry-biology-benchmarks.pdf.

When some inter-dependency exists between data cells, it is still possible to utilize the GPU and achieve significant improvements in performance, but typically the benefits will be less prominent than in completely independent cases. In programming terms, `map()` operations (such as multiplying each element by a factor, where the dimensions of the input and output are the same) tend to be more efficient on GPUs than `reduce()` operations (such as performing a group-by aggregation of the elements, where the dimension of the input exceeds the dimension of the output). Reduce-type operations might have to be reformulated to take maximum advantage of the parallelization to become truly efficient on GPUs.

Finally, the third aspect refers to a more technically detailed feature of GPU—the way calculation execution is organized. Without getting into too much technical detail, the performance of a GPU will be best if individual calculation cells follow the same (or very similar) linear path of execution. If the calculation contains many "`if-elseif-else`" statements, creating a lot of possible branches, execution performance will deteriorate. This is because certain groups of GPU cores must start and end their computations at the same time, and moreover, synchronization points for those cores must occur at the same point in the code. In the best case, this might result in some cores waiting for the ones with longer execution branches; in the worst case, the wait might be indefinite if some branches do not converge to the synchronization point. We take a look at the tip of the iceberg related to these more technical topics in the next section.

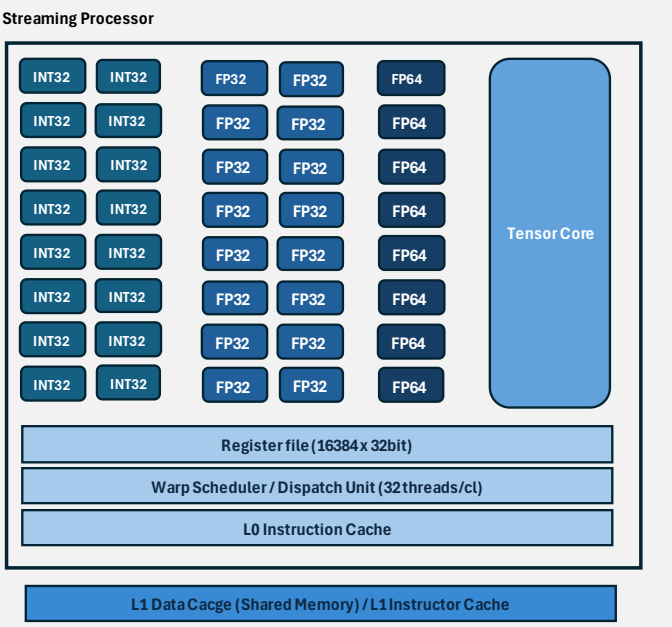
# GPU architecture fundamentals

Until now, we have only scratched the surface of the GPU architecture. In this section, we will take a slightly deeper look at this, as efficient use of GPUs for modeling requires an understanding of their architecture. This is by no means an in-depth or complete explanation, but it presents a few selected key concepts in a simplified way.[10]

As mentioned earlier, a single GPU consists of thousands of cores (in NVIDIA GPUs, also called 'CUDA cores'). These are physically organized on the GPU into streaming multiprocessors (SMs), subdivided into streaming processors (SPs). Cores in a single SP share control logic and in a single SM share a dedicated on-chip memory (called shared memory), significantly faster than the global off-chip VRAM. The number of SMs and cores per SM, as well as the amount of shared memory available in each SM, varies between different models and generations of GPUs.

As indicated previously, modern GPUs have even more types of cores present, specializing in different types of calculations. The exact types and numbers vary by GPU architecture and model. Figure 6 shows a simplified architecture of a single processor of an NVIDIA Ampere GPU (A100) streaming multiprocessor with (among others) 16 INT32, 16 FP32, 8 F64, and a tensor core for the aforementioned FMA matrix operations. A single SM in this architecture has 4 processors like the one depicted and includes some additional special functions units (SFU, e.g. for trigonometric function calculation) in each.

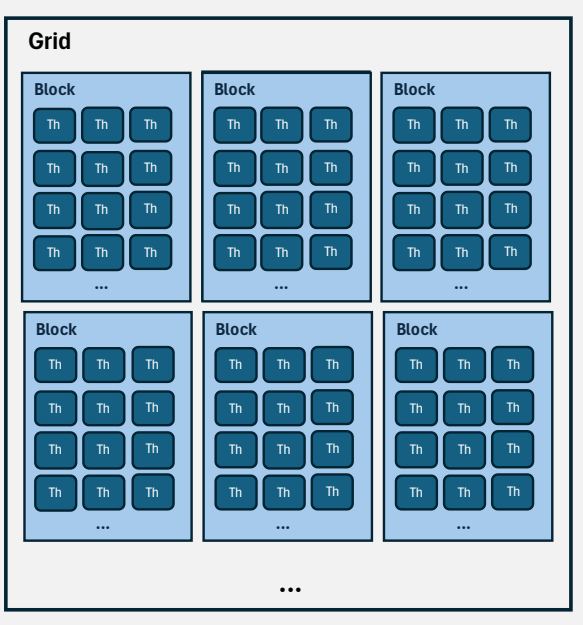
**FIGURE 6: NVIDIA A100 STREAMING PROCESSOR ARCHITECTURE (SIMPLIFIED)**




When a computation unit (also referred to as a *CUDA kernel*) is executed on a GPU, it is mapped to a set of fundamental GPU execution units—threads. Disregarding, for the sake of this explanation, more complex cases, we can assume that a single thread corresponds to a single CUDA core on a single data element on which computations are to be performed. These threads are organized into a thread grid and subdivided into thread blocks. The number of threads in a block and the number of blocks in a grid are parameters determined by the program invoking GPU computations. Based on these parameters and the hardware limitations of the specific GPU device used, thread blocks are allocated and queued to SMs for execution.

10. For a comprehensive overview, see Nvidia CUDA toolkit documentation. Retrieved December 3, 2025, from https://docs.nvidia.com/cuda/.

**FIGURE 7: ORGANIZATION OF THREADS INTO BLOCKS AND GRID**

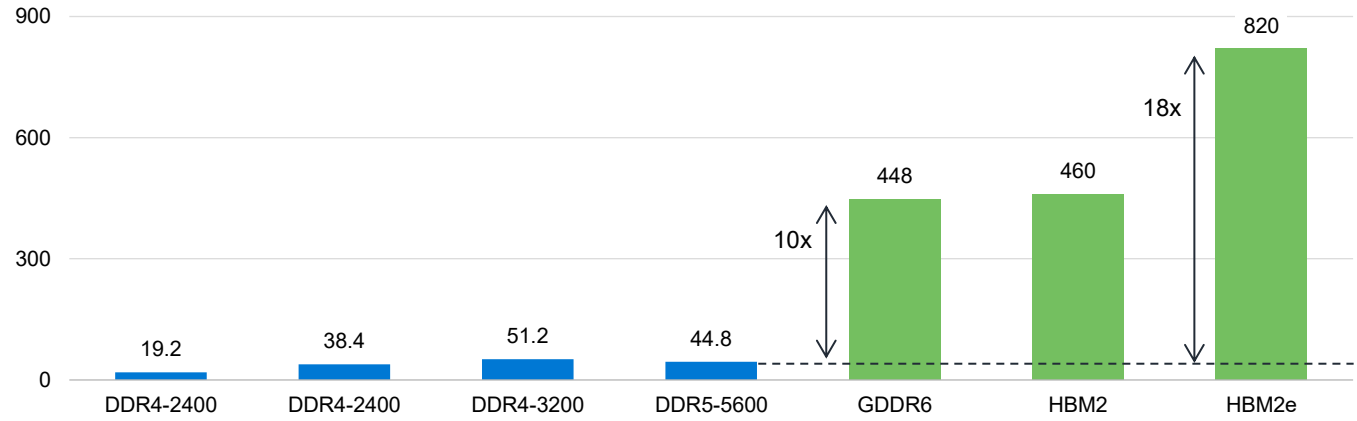



Subdivision into blocks is important because threads in the same block enjoy several privileges that threads from separate blocks do not, as they are executed on a single SM. First, threads within one block all have access to the same shared memory of the SM and can exchange information this way. Second, they can introduce a certain level of dependency between otherwise independent threads by being able to wait at a specific point in the code that calls for synchronization and to continue only after all of them have reached that point, a technique known as barrier synchronization. This allows all threads in the block to occasionally update shared information that could be used as input to the next stage of parallel calculations. Another element shared by the threads in one block is a register file, which is a dedicated, fast on-chip block of memory to store local variables for each thread.

An important concept in understanding parallelism on the GPU is a thread warp. This is the smallest group of threads that is executed simultaneously in an SM. For all NVIDIA GPUs to date, the size of this group has been set to 32 threads, but based on the documentation, it can be changed in the future. To achieve maximum performance, all threads in a warp can only execute the same instruction at any given time (although on different data elements). However, this becomes a constraint if the code execution in different threads diverges. Divergent threads in a single warp decrease efficiency, as those threads must wait for each other as different code branches are executed. This is the reason GPU code should be as uniform as possible across threads, and at the very least ensure that data elements with different operations (e.g., if-else branches) are not executed in the same warp.

Last, but not least, is the matter of memory use. Many types of memory must be considered when dealing with a GPU; not understanding the differences has the potential to significantly degrade performance of the code. The first major difference is the speed of the general-purpose computer RAM and the GPU VRAM. Nowadays, the former is typically a DDR5 type of memory, whereas in GPUs, this differs between GDDR6 (in consumer GPUs) and HBM2/HBM2e (in server GPUs). Even though RAM is customarily considered the fastest data source in a computer, as shown in Figure 8, GPU memory is roughly 10x–20x faster. This should clarify why managing the transfers between RAM and VRAM is such a critical area to optimize when aiming for high performance.

**FIGURE 8: MEMORY THROUGHPUT COMPARISON (GB/S)**




Data source: BittWare Article, Comparing DDR4 and DDR5 Memory. Bandwidth for FPGA Accelerator Cards[11][12]

11. BittWare. (n.d.). Comparing DDR4 and DDR5 memory bandwidth for FPGA accelerator cards. https://www.bittware.com/resources/ddr4-and-ddr5-performance-comparison/.

12 Double appearance of DDR4-2400 with different results is due to single-channel (lower result) and dual-channel (higher result) configuration.

However, the VRAM throughput of the global GPU memory is still considered very slow in GPU computation. Depending on the specific GPU considered, shared (and register) memory bandwidth can reach more than 12,000 Gb/s[13] (for Nvidia V100 GPU, newer GPUs have an even better performance), which is another 15x faster than VRAM. It is worth noting that the capacity of global memory in a GPU is usually measured in gigabytes, whereas each SM's shared (and register) memory is only several hundred kilobytes, so their memory usage models are very different.

As mentioned at the start of this section, this overview barely touches the tip of the iceberg representing GPU architecture, its constraints, and dependencies. For readers interested in a more extensive and technical description, we recommend more detailed material available, for example, in NVIDIA's CUDA Programming Guide. A thorough understanding of these concepts is a prerequisite for writing a truly performant GPU code.

# Actuarial problems and GPUs

There are a variety of computationally intensive problems actuaries encounter for which a GPU may offer performance improvements. The primary factors that determine the suitability of a problem for a GPU have been discussed. In this section, we will describe general types of problems in actuarial science and a variety of problems that we have implemented on a GPU.

### ASSET–LIABILITY MANAGEMENT

A broad category of modeling in the insurance industry is asset-liability management (ALM). The examples that follow illustrate the usage of a GPU to model assets, liabilities, or both. A more complicated aspect of ALM modeling is when there is an interdependence between assets and liabilities. There are several approaches to designing such a model. The two most common in Europe are full ALM (also called *dynamic*) and flexing-based ALM. The former relies on modelling liabilities and assets period by period and directly including the effects of one side on the other. That requires aggregation and allocation calculations in each period. As mentioned in an earlier section, this kind of reduce operation scales less efficiently for parallelization than simple *map* operations. This compounds the time-dimension interdependency already present in the liability cash flow projection modeling. This kind of model is also typically very memory-intensive, as it requires all liability and asset information to be kept in memory during all calculations.

Flexing approach, on the other hand, allows first projecting the deterministic base liability cashflows (e.g., before any impact from market-related factors, such as profit-sharing) for the whole projection period and then applying factor-based scaling (flexing) to these cashflows based on specific investment results from each stochastic economic scenario, period by period. This approach requires a sacrifice of some precision, as it is typically done at an aggregate level of liability cash flows (e.g., product/technical guarantee rate level), but is less memory- and computation-intensive. It also removes one of the dependencies and the need to aggregate–allocate at each step of the calculations, which makes it notably more suitable for GPU computations. A similar approach for assets could be considered, in which gross asset cash flows are pre-projected independently, aggregated to the granularity of an investment/disinvestment algorithm, and potentially scalar overlays such as defaults. Then, factor-based scaling is applied based on projected purchases, sales, and defaults.

The general problem with a complete ALM calculation chain for GPU computations is that, even with the flexing approach, there are still many very diverse calculations. This departure from uniform and linear flow across threads (in the same warp) is far from ideal and poses significant challenges to an efficient GPU implementation. It is, of course, *possible* to implement the complete flow on a GPU, or to select the most computationally intensive parts for GPU execution while keeping the higher-level flow on CPU. One could also try to reformulate the typical ALM approach to better fit the GPU paradigms.

For one of its clients in France, Milliman has built a simplified prototype ALM model supporting both flexing and full dynamic approaches and running both on CPU and GPU. In our benchmark results for the client, we saw around 10x performance increase over a comparable multi-CPU run (based on eight cores, 2,000 stochastic simulations, 100k policy portfolio, and 5,000 asset portfolio and flexing model). This was based on a very naïve CPU-to-GPU code conversion without any specific optimization around, for example, memory transfers or GPU architecture, as this time-boxed experiment was ancillary to primary project objectives. We also observed that for some parts of the calculations, performance improvements were significantly higher (reducing runtime by up to 95%, e.g., economic scenario data transformations or generating outputs), whereas for others, those improvements were much smaller (reducing runtime by only around 50%, e.g., aggregations of liability cashflows or setting up model structure).

13. Jia, Z., Maggioni, M., Smith, J., & Scarpazza, D. P. (2019, March 18). Dissecting the NVidia Turing T4 GPU via microbenchmarking. Citadel, 34. https://arxiv.org/pdf/1903.07486.

# Life liability products

A broad class of problems that fit the GPU model well is creating cash-flow projections of life insurance liabilities. If a company wants to do it at the most detailed level—policy or coverage—this will mean millions of policies projected over several decades. Such models can be quite heavy to run, even without involving stochastic scenarios (which could be applied, for example, to mortality or market data, and therefore increase both complexity and computational requirements). Even a simple liability cashflow projection model has one potential drawback for GPU conversion, as there is an interdependency within the projection along the time dimension (i.e., values at t are based on values at t-1), which, to some extent, limits the principle of independence of atomic calculations submitted to GPU threads.

We have a few examples of this type of modeling in subsequent sections of this paper. One can also explore a simplified case, such as this 2023 Milliman White Paper, "*Building a high-performance in-house life projection and ALM model: Architecture and implementation considerations in Python.*"[14] The focus of that paper was different, but it also shows a 20-fold increase in performance after translating the model into GPU-enabled code (for a sample size of 0.5 million policies).

### VARIABLE ANNUITIES

A Variable Annuity (VA) is a life insurance product, often with embedded options, that has exposure to the capital markets. Essentially, a VA is a tax-advantaged way to invest in the market, often with guarantees to protect the policyholder from downside risk.

Management of a block of VAs has high computational demands due to risk management and financial reporting requirements that require projecting cash flows for both risk-neutral and real-world simulations over many economic scenarios. The payoff profile of a VA is path-dependent, which creates interdependence in the time dimension, limiting that aspect of parallelism.

Here are some examples of the types of metrics companies may need to run:

- Risk-neutral valuations for market sensitivities for hedging, pricing, and/or reserving
- Real-world simulations for statutory reporting and capital calculations
- Stochastic-on-stochastic simulations to test hedging strategies, run plan scenarios, and/or generate results of a clearly-defined hedge strategy (CDHS) for financial reporting

VA is a separate account product that typically has little general account exposure at the contract level, resulting in minimal interdependence between contracts for most types of runs. As such, the data set is highly parallelizable for many of the intense computational workloads that insurers need to generate. We have built GPU models for stochastic-on-stochastic, risk management, hedge simulation, and financial reporting in both C++ and Python with Numba.

### FIXED INDEX ANNUITIES

The Fixed Index Annuity (FIA) is another type of annuity in the life insurance space that has capital markets exposure. As with the VA, the FIA has path dependence and metrics that require projection of liability cash flows. Historically, risk management of these products was the primary computationally intensive metric, but the advent of VM-22, a U.S. principles-based reserving framework requiring valuation under stochastic real-world scenarios, has added additional areas of heavy computational demand.

Unlike VA, FIA is a general account product that may add an additional layer of complexity from a modeling perspective, discussed in the ALM section above. For some types of runs, the evolution of the general account in aggregate is necessary to capture at a policy level. This additional layer of dependence between the contracts reduces parallelism. The degree to which this may impact GPU throughput will be dependent on the number of contracts and/or scenarios being run and the number of times the calculation requires synchronization across contracts.

Our experience modeling FIA on GPUs has been focused on risk management and hedging using models built in C++. For this paper, we built a toy model in Python/Numba to simulate pricing an FIA with a Guaranteed Lifetime Withdrawal Benefit (GLWB) in the VM-22 framework.

The model is not fully developed from an assumptions and product feature perspective, but is a means of looking at relative impacts of modeling decisions and constraints. All FIA simulations are for 50 years at a monthly frequency over 1,000 real-world scenarios. The model was run on a CPU and GPU using Python with Numba.

Several observations from this exercise illustrate points highlighted elsewhere in this paper. In particular, this exercise gives a sense of how the design of the implementation may change as the problem evolves, the impact of memory layout, and how reducing parallelism can impact performance.

14. Maciejewski, K., Echchelh, M., & Sznajder, D. (2023, March 13). Building a high-performance in-house life projection and ALM model: Architecture and implementation considerations in Python. Milliman. https://www.milliman.com/en/insight/building-in-house-projection-alm-model-python.

The runtime comparisons that follow are based on this model and are on machines from Azure that are running the Linux operating system:

- Standard_HB176rs_v4, a high-performance CPU VM with 176 CPU cores priced at $7.20 per hour
- Standard_NC80adis_H100_v5, sporting two H100 GPUs priced at $13.96 per hour

All runs are double precision (float64) and do not include input/output or compile time in the runtime. The GPU execution time includes memory transfer to and from the GPU. It should be noted that this model is only utilizing only one GPU since it is not equipped to utilize multiple GPUs.

Figure 9 shows results under the assumption that the contracts have no interdependence. In other words, each policy is completely independent of any other policy. With that in mind, this run is a single kernel call to the GPU to calculate all policies, scenarios, and timesteps.

**FIGURE 9: FIA PRICING BENCHMARK CPU TO GPU COMPARISON (EXECUTION TIME)**

| POLICIES | CPU (172 CORES) | GPU (1 GPU) | RATIO |
|---|---|---|---|
| **32** | 0.05 | 0.01 | 3.98 |
| **128** | 0.14 | 0.02 | 6.47 |
| **1280** | 1.03 | 0.12 | 8.52 |
| **20480** | 15.91 | 2.67 | 5.95 |
| **81920** | 63.82 | 10.24 | 6 |

The results demonstrate that even for a small number of policies, the GPU shows a runtime improvement over the CPU. Although the number of policies is small, each policy accounts for 600,000 iterations through the logic (scenarios × timesteps). The CPU and GPU both exhibit a nonlinear response to the increase in policies initially since neither is saturated with a small number of policies. As the number of policies increases, both the CPU and GPU timings scale approximately linearly and approach a sixfold relative performance improvement of the GPU relative to the CPU.

The next part of the exercise is to consider what happens if there is dependence between the policies. This is the type of complication that one would encounter when simulating dynamic asset modeling, as described in the ALM section. To accomplish this, we will run the kernel multiple times to move the policy information along each scenario for a specific number of timesteps, and then return control to the CPU. We want to store the state of each policy by scenario on the GPU to avoid unnecessary data transfers.

The implementation of the above model loaded a single representation of the policy data in global memory. For this exercise, however, we need to store each policy by scenario to capture the policy state between kernel calls. In the initial version, the model reads the policy data from global memory and loads it into local data structures. This design reduces the number of calls to global memory, but to accommodate this exercise would necessitate an explicit update of the information in global memory to capture the state when the kernel cedes control back to the CPU. To avoid an explicit update and repeated loading into the local memory structures, the model was adjusted to operate directly on data from global memory. A comparison of the original implementation to the new version is shown in Figure 10.

**FIGURE 10: FIA PRICING BENCHMARK IMPLEMENTATION COMPARISON (EXECUTION TIME)**

| POLICIES | ORIGINAL | ALTERNATIVE | PERCENT INCREASE |
|---|---|---|---|
| **32** | 0.01 | 0.09 | 612% |
| **128** | 0.02 | 0.20 | 848% |
| **1280** | 0.12 | 1.82 | 1405% |
| **20480** | 2.67 | 28.74 | 975% |
| **81920** | 10.24 | 114.79 | 1021% |

The impact of the implementation change was an increase in runtime by almost 11 times. This reduction in performance is likely tied to two related things. The new model has many more calls to global memory than the previous version, and the memory layout in global memory for the policy data structure is not implemented in a way to facilitate coalescing of memory. It is possible to alter the way the policy data is stored and used, but the resulting implementation would be tedious and less maintainable.

The final version of the model for this exercise was a hybrid of the original and proposed models. We reverted to utilizing local storage to minimize calls to global memory and added logic to update an expanded representation of the policy data in global memory. The result was performance in line with the initial version of the model.

With this new version in place, we returned to the exercise and ran the model with varying frequencies of checkpoints. The label “Frequency” in Figure 11 is the frequency at which we returned control to the CPU. At the end of each call to the kernel, we copied the entire results data structure back to the CPU to emulate possible methods for implementing the asset strategy. We moved the results from the GPU back to the CPU to account for the possibility that the CPU would run the asset strategy. If the asset strategy were also run on the GPU, there would be no need to move the results to the CPU. Figure 11 shows the percentage increase in runtime when simulating the flow of control at varying frequencies to run an asset strategy.

**FIGURE 11: FIA PRICING BENCHMARK CPU CHECKPOINT FREQUENCY (EXECUTION TIME)**

| FREQUENCY | TIME (S) | PERCENT INCREASE |
|---|---|---|
| **None** | 14.59 | 0% |
| **Annual** | 58.18 | 299% |
| **Quarterly** | 85.53 | 486% |
| **Monthly** | 123.47 | 746% |

The reduction in performance is from the transfer of data back to the CPU and the reloading of data from global memory to the local data structures. The impact of the latter would be reduced by the restructure of the policy data contemplated above.

### ASSET VALUATIONS AND HEDGING

Another example from the area of asset–liability and risk management is the valuation of assets in an insurance company's portfolio. Nowadays, companies perform a quickly increasing number of projections for various purposes:

- Best-estimate calculations for products with financial profit-sharing
- Risk-based capital calculations
- Sensitivities for hedging and capital optimizations

For this paper, we have developed a simple engine for bond market valuation using the discounted cash flows approach. It can be shown that this problem fits the criteria of GPU-suitability very well.

Typical bond portfolios held by life insurance companies vary from several hundred to thousands of individual instruments, with more added in each projected reinvestment cycle to reflect the simulated purchases. Each bond needs to be valued at each projection step until its maturity and typically across a number of stochastic scenarios. Therefore, the dimensionality of the problem (data size) grows quickly: `[bonds x projection steps x scenarios]`.

Valuation of each cell of this array can be considered independent of the others. One might argue (and rightfully so) that, at least in some cases, fixed-rate bond cash flows from one projection step to another remain largely unchanged and can be reused. This can be considered, however, one case for which the benefit of independence is bigger than the efficiency of performing each calculation exactly once. The simplicity of the implementation and performance gain outweigh the need to repeat the cash flow calculation.

The third key element—code branching—is also limited. Each bond has a different maturity term, so although the exact number of computations in each cell varies, the logic of the calculation flow remains uniform and linear.

In the valuation engine developed for this paper, we have made several simplifying assumptions, which facilitated the development without impacting the observations and conclusions:

- We used fixed-coupon bonds only
- We generated random discount factors
- We performed valuations annually from start of the projection until maturity of the last bond in the portfolio

For the benchmark exercise, we used a dummy bond portfolio with an average term-to-redemption of 20 years (maximum 40 years, minimum 1 year), a fixed coupon payment, and varying coupon payment frequencies (from monthly to annual). The sample size mentioned in the following figures refers to the (rounded) total number of market value calculations (equal to the product of three dimensions: number of bonds, number of stochastic simulations, and number of years until maturity for each bond). As an example, the 10-million sample corresponds to 550 bonds, 960 simulations, and an average of 19.4 years until maturity (giving a total of 10,243,200 calculations).

The implementation was done in Python using Numba (for both CPU and GPU variants), using single precision (float32) and Fortran-ordering of array elements in memory (which, in our implementation, facilitated splitting arrays in the multi-GPU context).

For the main part of benchmarking this use case, we used a machine with an EPYC 7402P CPU, 256GB RAM, and 4x NVIDIA Tesla P100s with 16GB VRAM each. For the test runs, we used different combinations of CPU cores and GPUs, as well as various sample sizes. The P100 is a relatively old generation of GPU (based on the Tesla Pascal architecture, introduced in 2016), so we also include the computation time for a modern H100 GPU with 80GB VRAM, provisioned from a specialized GPU-cloud instance provider.[15]

We timed the execution on the host side using the standard Python time library, where relevant, and, where possible, split the total execution into loading data into VRAM, pure computations, and retrieving results from VRAM. Loading data from disk and generating discount factors are not included in the results.

15. See the RunPod homepage page at https://www.runpod.io.

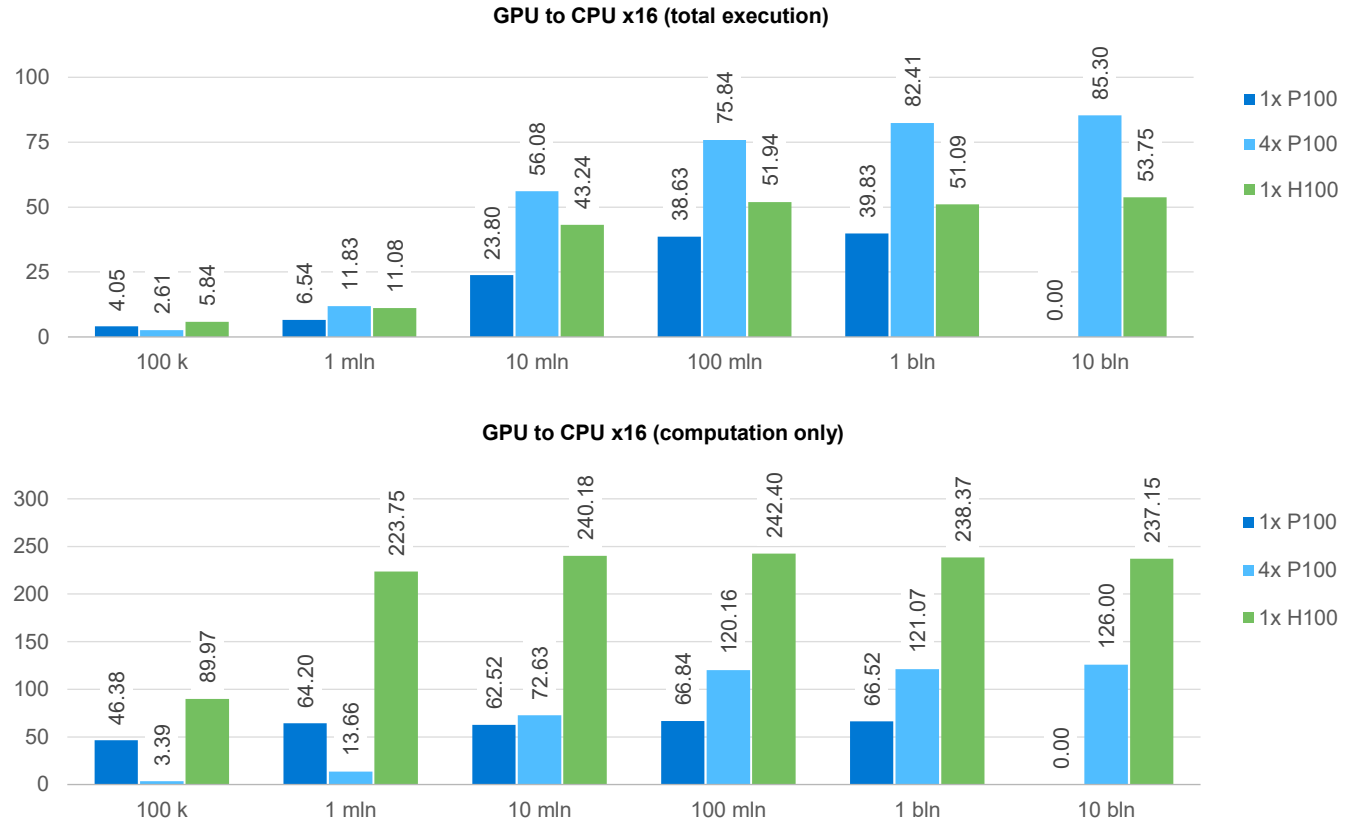
**FIGURE 12: BOND VALUATION BENCHMARK (TOTAL EXECUTION TIME IN SECONDS)**

| SAMPLE SIZE | 1 K | 10 K | 100 K | 1 MLN | 10 MLN | 100 MLN | 1 BLN | 10 BLN[16] |
|---|---|---|---|---|---|---|---|---|
| **CPU: 1 core** | 0.0012 | 0.0097 | 0.0862 | 0.8518 | 8.5158 | 85.0788 | 849.2658 | 8562.08 |
| **CPU: 16 cores** | 0.0028 | 0.0030 | 0.0226 | 0.1891 | 1.5362 | 15.0580 | 147.3351 | 1462.8326 |
| **GPU: 1x P100** | 0.0048 | 0.0069 | 0.0094 | 0.0429 | 0.0753 | 0.4087 | 3.8794 | N/A[17] |
| **GPU: 4x P100** | 0.0098 | 0.0090 | 0.0082 | 0.0151 | 0.0439 | 0.2114 | 1.6167 | 16.1857 |
| **GPU: 1x H100** | 0.0019 | 0.0028 | 0.0039 | 0.0171 | 0.0355 | 0.2899 | 2.8836 | 27.2149 |

**FIGURE 13: BOND VALUATION BENCHMARK (COMPUTATION ONLY, TIME IN SECONDS)**

| SAMPLE SIZE | 1 K | 10 K | 100 K | 1 MLN | 10 MLN | 100 MLN | 1 BLN | 10 BLN |
|---|---|---|---|---|---|---|---|---|
| **CPU: 1 core** | 0.0012 | 0.0097 | 0.0862 | 0.8518 | 8.5158 | 85.0788 | 849.2658 | 8562.08 |
| **CPU: 16 cores** | 0.0028 | 0.0030 | 0.0226 | 0.1891 | 1.5362 | 15.0580 | 147.3351 | 1462.8326 |
| **GPU: 1x P100** | 0.0007 | 0.0002 | 0.0005 | 0.0029 | 0.0245 | 0.2285 | 2.2136 | N/A |
| **GPU: 4x P100** | 0.0044 | 0.0015 | 0.0029 | 0.0025 | 0.0236 | 0.1213 | 0.8373 | 9.7637 |
| **GPU: 1x H100** | 0.0003 | 0.0003 | 0.0003 | 0.0008 | 0.0064 | 0.0621 | 0.6181 | 6.1683 |

**FIGURE 14: PERFORMANCE IMPROVEMENT GPU TO CPU X16**




16. Due to very high volatility of the time required to initialize the GPU and return control to the CPU, the number in the last row of this column has been calculated as a proxy using actual memory transfer times and calculation time, and an average overhead from 1 billion sample runs.

17. A single P100 GPU with 16 GB VRAM is not able to handle that much data.

Figure 12 presents an overview of the results in terms of total execution time, and Figure 13 presents the results in terms of pure computation time (which is the same in the case of CPU-based benchmark; we excluded the disk to general memory loading time, as it is the same regardless of CPU or GPU being used. It is worth noting that in the case of computing resources provisioned in the cloud, these fixed overhead times, including the time it takes to boot up and prepare a virtual machine for calculation, will be more expensive for the GPU machine, given that it would typically have a higher cost per unit of VM uptime.

Figure 14 shows comparative graphs of performance improvement factor between various GPU runs and the corresponding 16-core CPU run, which we believe is more representative and realistic than running on a single core. From this overview, we can clearly see that for small samples, the benefits of using a GPU are small, as the overhead related to the GPU initialization and use overshadows the pure computational benefit. For this model, it seems a sample size between 10 and 100 million bond valuations is required to realize stable execution times, reflecting a performance improvement of approximately 40x for the older P100 GPU and 50x for the newer H100. However, when looking at pure computational time, excluding data transfer times, we see that at only approximately 1 million calculation cells, a stable level of around 65x improvement is reached on the P100 GPU and an impressive factor of 240x on the H100 GPU.

In a multi-GPU context, it is important to mention that there are different ways multiple devices can be utilized to distribute calculations. The approach used in this benchmark was to create separate CPU threads (using the standard Python threads library), one for each GPU device, and let each thread manage the transfer of its relevant chunk of data, the CUDA kernel invocation, and the transfer of the results back to general RAM. The final step was to put the partial results together into a single array to align the final output with the single-GPU and -CPU cases.

The benefits observed when using multiple GPUs were higher, as expected, but did not demonstrate perfectly-linear scaling with the number of GPUs[18], suggesting an increase in the overhead when working with several devices. In terms of total execution, a stable point seems to be achieved between 100 million and 1 billion cells (25 million and 250 million per GPU) at an improvement factor of around 80x. In terms of pure computation, there is apparently a bit more stability, with the improvement factor increasing to reach 120x for 100 million samples.

Keeping in mind that these are factors of improvement over a 16 core CPU run, and even taking into account the time spent on data transfers between memory types, these results still show an very impressive performance gain that would bring a model runtime of 1 hour to just over 1.2 minutes given a single modern GPU (and 1.5 minute given a cheaper, older GPU). Achieving the same result using a CPU grid would require utilizing almost 600 cores!

Using multiple GPUs in parallel will likely be overkill for most actuarial workloads, but it can help when the data size exceeds the capability of a single GPU (as was the case with the 10-billion-sample dataset in this benchmark). Even then, an alternative is to split the data into chunks and submit the calculations to the same GPU in a sequence. Moreover, with modern server-grade GPUs having memory of 100+ GB, memory exhaustion on a single device by actuarial model data is unlikely.

The implementation of this benchmark case was done in a relatively “naïve” way, as far as GPU optimizations are concerned. As we will discuss in the next section, there is potential for significant improvement if GPU architecture-specific details are taken into account while fine-tuning the code. Similarly, CPU performance could be enhanced using optimizations targeted at specific hardware topology and microarchitectural elements of high-performance machines. Each requires a substantial amount of highly technical knowledge, which is likely unknown to even the most tech-savvy actuaries. Therefore, we see this simple approach as more realistic for an actuarial modelling use case.

# Efficient GPU modeling

The technical expertise required to utilize GPUs has diminished over time with advances in CUDA and with the accessibility of CUDA through high-level languages. The need for advanced technical expertise to efficiently utilize a GPU persists for many actuarial applications. The value added from tuning a GPU falls on a spectrum that is dependent on the problem. There are instances where converting a CPU implementation of a model to a GPU implementation is straightforward, and tuning may only result in modest performance gains. On the other end of the spectrum, tuning may deliver substantial gains in performance and cost reductions. Some aspects of utilizing a GPU efficiently are related to understanding the structure and mechanics of a GPU processor. There are other aspects of tuning that center around how to best implement a calculation to optimize parallelism for a specific use case. There are instances where the optimal way to use a GPU is not optimal for a CPU.

18. A point worth noting—this is also in line with what has been observed when increasing the CPU core count.

Different levels of understanding the technical aspects of a GPU architecture enable several relatively "standard" optimizations that are widespread in more sophisticated GPU use cases. The simplest example is choosing the right shape of the thread grid—in particular, the number of threads per block. Recall from our earlier discussion on thread warps that 32 is the minimum number of threads that can be executed simultaneously in a block. Therefore, best practice is to use a multiple of 32 as the number of threads in a block, thereby eliminating "empty" threads. Additionally, each GPU imposes a maximum on the number of threads per block (typically 1,024 or 2,048). Clearly, these two constraints still leave a wide range of possible warp sizes.

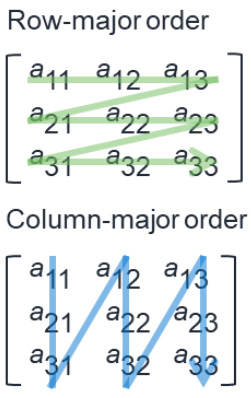
There is no single recipe for an optimal number; depending on the computational problem at hand and the specifications of the GPU used, different values might be the best in different cases. The relevant individual GPU specifications include, among others, the maximum number of blocks per SM, the maximum number of threads per SM, and the amounts of shared and register memory per SM. Finding the optimal choice is to seek a balance between *occupancy* maximization and *memory bandwidth* optimization. Occupancy is a measure of how many threads (or warps) are active in an SM at any given time, compared to the theoretical limit. This is important because it keeps the utilization of the SMs high and allows for *latency hiding*, which is a technique that allows the GPUs to quickly switch from threads that are stalled (waiting, e.g., for memory operation) to others that can perform their operations and, in that way, minimize the effect of some bottlenecks. On the other hand, there are limits on shared and register memory sizes on each SM. The more threads are executed on an SM, the less memory is available per thread, which might increase latency due to additional global memory access required.

Another way to minimize latency and increase performance is to use *streams*. They allow for a certain degree of asynchronous operations on the GPU—simultaneously executing kernel invocations and memory transfers between the host and GPU, as well as independent kernel executions to run in parallel. Minimizing bottlenecks and ensuring maximum GPU load can ultimately be quite a challenging task. Fortunately, there are sophisticated tools, such as NVIDIA Nsight Compute[19] for CUDA workflows, that can analyze workload performance and generate guidance on how to potentially optimize it.

It is also important to understand the GPU features that are available on the device one is using and how those features can be accessed in different programming languages and their libraries. Some will allow greater control over technical execution parameters, and some provide additional tools. An example of this can be GPU Direct Storage (GDS),[20] which is a toolkit that enables direct data transfers between the GPU and data storage. This allows skipping the CPU and general system RAM when reading and writing files, significantly speeding up input-output operations. In the aforementioned example of a prototype ALM model built for a French Milliman client, we saw a 12x decrease in the time required to generate result files after enabling Direct Storage for this step.

A prime example of a more complex optimization is *memory coalescing*. When different threads of the same warp request data from the global GPU memory, the time it takes the GPU to fulfill those requests depends on where the data are physically located in memory. In the best case, requested data cells are adjacent to each other, and, as a consequence, can be provisioned back to threads all at once. However, if those cells are spread in different parts of memory, it will take several cycles to collect all of them and ultimately fulfill the warp request. *Memory coalescing* ensures threads within the same warp access data that are adjacent in global memory. When one is working with multidimensional numerical arrays (as is typically the case in the GPU context), ensuring the right alignment between threads and data cells in the array boils down to understanding the C-style versus Fortran-style memory order and accessing the arrays accordingly. In C-style convention, 2D arrays are stored in memory in row-major order, meaning elements of a single row are next to each other in memory. In Fortran-style convention, things are reversed, and column order is used, putting elements of the same column next to each other in memory. In higher dimensions, that arrangement corresponds to the innermost index changing most quickly (row-major memory representation) and the outermost index changing most quickly (column-major memory representation).

**FIGURE 15: ROW- VERSUS COLUMN-MAJOR MEMORY ORDERING**




19. See the NVIDIA Nsight Compute product page at https://developer.nvidia.com/nsight-compute.

20. See the NVIDIA GPUDirect Storage product page at https://docs.nvidia.com/gpudirect-storage/.

Another memory-related example would be optimizing the code to use shared memory (or registers) instead of global memory. As shown before, the latter is the slowest of the available GPU memory types, so if different threads of the same thread block need to use specific data elements multiple times, it is more efficient to make one thread copy those elements to shared memory so that the other threads can access them there. Given the very limited amount of shared memory, this must be carefully designed to achieve the expected improvements. This refinement can then be expanded to optimize which threads access which physical banks of the shared on-chip memory. Specifically, when multiple threads attempt to access memory in the same bank, they can block each other, delaying the execution. Managing access to shared memory so that simultaneous requests are served by different memory banks removes this bottleneck. There are many examples of such low-level technical optimizations available on the Internet. One of them, which uses an example of a reduction-type problem—namely, summation—was presented in an Nvidia webinar some years ago: Optimizing Parallel Reduction in CUDA21. In this webinar, the author shows how he achieved a total 30x performance improvement in his final optimized kernel over the initial naïve one, proving how important this step is if one is after the lowest runtimes possible.

On the other end of the spectrum, there are things that do not require deep knowledge of GPU architecture, but more of an algorithmic view of how massively parallel processing is different from "typical" sequential computing. That means that very different algorithms might be significantly more efficient on GPUs than those taught and used in conventional CPU programming. The most basic example for this is substituting a "main" loop of the sequential solution with the distribution to threads in a GPU context. There are many more possibilities, and sometimes finding an efficient algorithm to solve a problem on a GPU requires out-of-the-box thinking. For example, in some cases it might be more efficient overall to do things that seem counterintuitive—such as avoiding a return of control to the CPU to perform calculations more efficiently executed there and instead embedding those calculations inside an existing GPU kernel—to avoid memory transfers. A similar technique is to repeat a calculation that uses identical data and operations in all threads, which, in a CPU implementation, might be better to perform once and cache to realize the requirements of independence, or to repeat calculations within a single thread that could otherwise be cached to overcome memory limitations.

This section barely touches on the various ways in which GPU code can be made more efficient. We invite the more curious readers, not afraid of technicalities, to research these topics further online (particularly using a rich NVIDIA blog and webinar library). It cannot be overestimated how important this is, given examples showing that benefits from optimizing might even outgrow benefits from the naïve conversion from CPU to GPU in some cases.

## Cost–Performance analysis

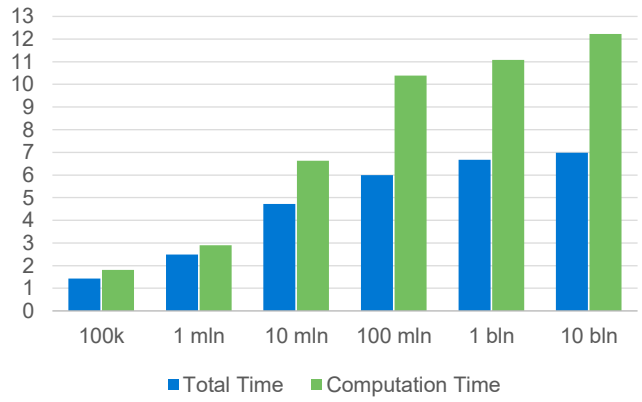
Based on the benchmarks presented in this paper, it is clear that even a single GPU can significantly outperform multi-core CPUs, with the exact performance improvement factor heavily dependent on the type of problem, hardware used, and GPU-specific optimizations applied to the algorithm and implementation.

The key question remaining is how much does this performance cost? As we mentioned earlier, due to the AI hype, GPU prices soared in recent years. But their availability in different cloud offerings has also considerably increased. GPU instances are available for ad hoc and long-term provisioning from all the main cloud providers (such as Azure, AWS, GCS).

As a first step in evaluating the relative cost-performance ratio, we consider recent generations of CPU and GPU instances provisioned in Azure with the Linux operating system and using the bond valuation benchmark performance ratios as the basis:

- Standard_HB176rs_v4, a high-performance CPU VM with 176 CPU cores priced at $7.20 per hour
- Standard_NC80adis_H100_v5, sporting two H100 GPUs priced at $13.96 per hour

**FIGURE 16: AZURE COST-ADJUSTED PERFORMANCE RATIO GPU (2X) TO CPU (176X)**




21 Harris, M. (n.d.). Optimizing parallel reduction in CUDA [NVIDIA Developer Technology]. NVIDIA. https://developer.download.nvidia.com/compute/DevZone/C/html/C/src/reduction/doc/reduction.pdf.

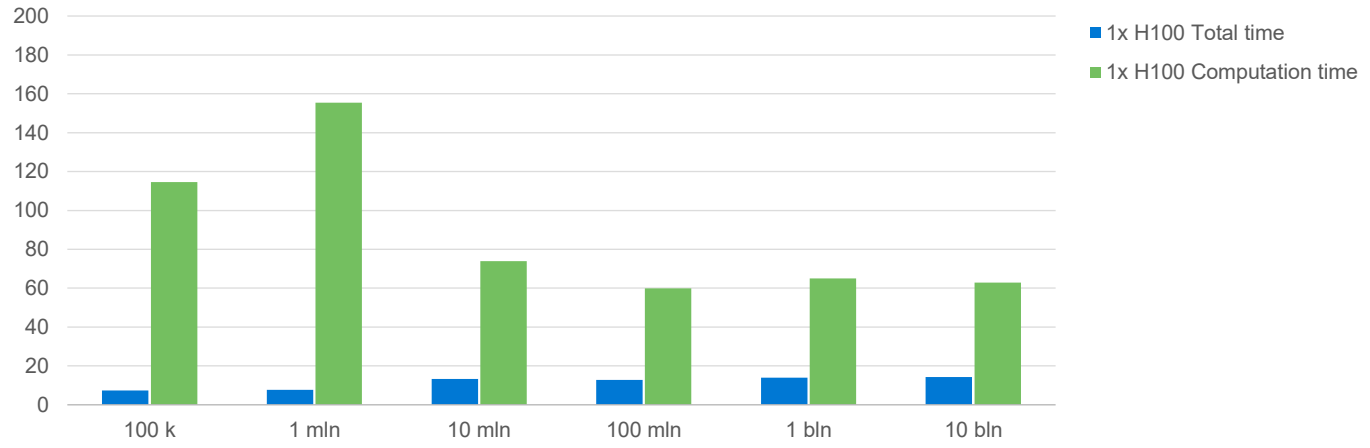
At least in Azure, after accounting for cost differences, the benefit of GPUs over CPUs is more muted, though still substantial, approximately a single order of magnitude. Thus, demonstrating that when determining the potential cost savings of converting a model from a CPU to a GPU, using the computational improvement alone (potentially two or three orders of magnitude better than a single CPU) may overestimate the monetary savings by an order of magnitude or more.

As indicated, there are also companies specialized in provisioning GPU compute power, which, on one hand, provide a wider variety of GPU models and full control over which GPU VM one might want to use, and on the other, offer more competitive prices for the GPU VMs than generic providers such as Azure or AWS. An example of such a specialty provider is the aforementioned RunPod.IO[22] used in one of our benchmarks. Below, we present the cost-adjusted performance factor for the H100 GPU provisioned from RunPod.IO relative to the 176-core CPU instance provisioned from Azure. The cost of the Azure instance with 176 cores (Standard_HB176rs_v4) at the moment of executing this benchmark was $7.20 per hour, whereas the cost of using a RunPod.IO single H100 PCIe GPU instance was $1.99 per hour.

Factoring in the cost ratio into the performance ratio, we can clearly see in Figure 16 that the GPU is not only faster but also cheaper to use, resulting in stable and impressive total cost-adjusted performance improvement factors (note that the pure performance comparison was done against a 16x core instance, whereas here it is against a 176x core instance).

As most companies might be bound by their IT policies, long-term contracts, and infrastructure choices to use a specific cloud service provider, the cost-adjusted GPU benefits might be somewhat less prominent (however, still clearly visible and tangible). If using a dedicated provider specialized in GPU cloud services is an option, it should definitely be considered, as their pricing tends to be more competitive.

**FIGURE 17: RUNPOD.IO COST-ADJUSTED PERFORMANCE RATIO GPU (1X) TO CPU (176X)**




22. Accessed 12 December 2025: https://www.runpod.io.

## Conclusions

The inclusion of CUDA in higher-level programming languages has reduced the expertise needed to migrate a model from a CPU implementation to a GPU implementation. The ability to effectively tune a GPU may still necessitate a higher level of technical expertise, depending on the problem. Furthermore, the capacity to exploit GPU hardware is also different between the languages. Use of CUDA or OpenCL from C/C++, for example, enables explicit access to GPU hardware, but requires extensive programming expertise. Conversely, languages such as Python (with Numba) and Julia or Mojo require only modest programmer effort to leverage the GPU, but do not expose the same capacity for fine-tuning. Although large performance differences are possible depending on the language used and the GPU-specific optimizations applied, even in the most naïve approaches enabled by Python, the model runtime improvements from enabling GPU computations can be impressive and, at the same time, more cost-efficient than scaling up cores for the CPU models.

There is also an increasing number of proprietary modeling software that explores or includes the possibility of leveraging GPU computational capabilities in one way or another. Typically, an attempt to generically convert any model code into GPU code will be less efficient than designing a dedicated algorithm and implementation for a specific problem. This, in turn, can typically be significantly further improved by applying GPU-specific technical optimizations. In any case, for models that meet the broadly outlined GPU-suitability criteria and are computationally demanding, a GPU seems to be an excellent path to one- or two-orders-of-magnitude performance improvements, with somewhat lower, but still significant, cost savings versus CPU-based approaches.

## Solutions for a world at risk™

Milliman leverages deep expertise, actuarial rigor, and advanced technology to develop solutions for a world at risk. We help clients in the public and private sectors navigate urgent, complex challenges—from extreme weather and market volatility to financial insecurity and rising health costs—so they can meet their business, financial, and social objectives. Our solutions encompass insurance, financial services, healthcare, life sciences, and employee benefits. Founded in 1947, Milliman is an independent firm with offices in major cities around the globe.

**milliman.com**

**CONTACT**

Karol Maciejewski
karol.maciejewski@milliman.com

Chad Schuster
chad.schuster@milliman.com

Jim Brackett
jim.brackett@milliman.com

Corey Grigg
corey.grigg@milliman.com

Milliman