A Proposal for Improved Computing in Actuarial Science

Introduction

When reading the Chip War by Chris Miller (2022), a book discussing the geopolitical and economic ramifications of microchips, my natural inclination was to connect microchips to actuarial science. Throughout the book, Miller discusses how microchips power all electronic products and even power artificial intelligence ("AI"). One specific microchip, graphic processing units ("GPU"), are particularly key for AI because GPUs drastically reduce the time needed to calibrate (or "train") AI models (Miller, 2022). However, artificial intelligence will not be my focus but rather the GPUs. GPUs speed up processing times because they are more effective at parallel processing than computational processing units ("CPU") (Miller, 2022). Now you might be wondering, what is parallel processing? Let's start with its inverse. Traditionally, computers would sequentially complete each task (task 1, then task 2, then task 3, etc.). The problem here is if tasks 1, 2, … and n are independent of one another, then this approach wastes computer resources. On the other hand, parallel processing calculates each task simultaneously by running multiple threads[1] at the same time. Each CPU core[2] is limited to two threads so the maximum number of the threads that can be utilized by the CPU is 2 times the number of cores (Soyata, 2018). For GPUs, each GPU core can have more than 32 threads and the number of cores in a GPU can number in the thousands (Soyata, 2018). I must add that there is a limitation to GPUs. The memory per thread is naturally lower for GPUs compared to CPUs based on the sheer number of threads in GPUs. However, in a later section, I will discuss overcoming this obstacle.

Now that we have covered the computer science, let us discuss the actuarial science. Actuaries run models that project future income statement items and balance sheet lines for blocks of insurance liabilities. We will call these models "actuarial liability models" or "liability projection models." My argument is that actuarial liability models can benefit from parallel computing. In my experience actuarial projection models already have some sort of parallel processing. However, these model uses CPU-based parallel processing as opposed to GPU-based. Currently, the actuary sends their model to the cloud and the cloud (i.e., a data center typically owned by Microsoft, Amazon, or Google (Richter, 2023)) allocates resources to perform the calculations. However, CPUs are limited with fewer threads than GPUs. With careful memory management, I believe that GPU-based parallel computing could be applied to liability projection models. This will allow for quicker turnaround actuarial work products and allow actuaries to spend more time thinking, rather than running models. The remainder of this paper will be structured as follows;

- I will first present a simplified actuarial liability model in the Actuarial Liability Modeling Framework section.
- In the next section, I apply parallel processing to actuarial models.
- Lastly, I will discuss memory management which will be the challenge with GPU parallel computing.

[1] Think of each thread as a computer's "worker" that performs calculations.
[2] A core is the computer's processor and is responsible for reading instructions and allocating computing resources (threads in footnote (1) above) to the task (Harding, 2022).

Actuarial Liability Modeling Framework

An actuarial liability model estimates the future cash flows and balances for a group of similar insurance policies (often called a block). It does this by adding together the expected cash flows and balances for each individual cell.[3] Since this is an actuarial liability model, we are dealing with the projection of liabilities (what the insurance company owes) rather than its assets. A cell is a group of similar insurance policies based on policy features, age, sex, risk class, etc. Actuarial models rely on the concept of expected value. By adding hundreds of thousands expected values together, it is assumed that these projections will be reliable in aggregate. The key to reliability is the underlying assumptions which are used to calculate probabilities.

For the remainder of this paper, I will focus on the projection of premium income, benefits, per policy expenses, premium taxes and reserves. There are other cash flows but I am limiting my analysis for simplicity. To further simplify, my model will be for life insurance where the only decrement[4] is mortality (no lapses for simplicity[5]). The equations below, summarize the calculations;

Definitions

$${}_tq_x^r = \ Probability\ someone\ aged\ x, of\ risk\ class\ r\ dies\ by\ time\ t$$

$${}_tP_x^r = \ Probability\ someone\ aged\ x, of\ risk\ class\ r\ lives\ till\ time\ t$$

$$i = Individual\ insured\ person$$

$$x(i) = Age\ of\ insured\ i$$

$$r(i) = Risk\ class\ of\ insured\ i$$

Equations

1. $Premium(t,i) = Premium\ Rate(t,i) * \ {}_tP_{x(i)}^{r(i)}$
2. $Benefits(t,i) = Policy\ Benefit(t,i) * \ {}_{t-1}P_{x(i)}^{r(i)} * \ {}_1q_{x(i)+t-1}^{r(i)}$
3. $Premium\ Tax(t,i) = Premium\ Tax\ Rate * \ Premium(t,i)$
4. $Per\ Policy\ Expense\ (t,i) = Per\ Policy\ Expense(t,i) * \ {}_tP_{x(i)}^{r(i)}$
5. $Reserve(i,s) = PV\big(Benefits(i)\big) - PV\big(Premiums(i)\big)$[6] $over\ all\ t > s$

Applying Computational Theory

In this section we will analyze the equations above from a computer programming view. I have following observations that are relevant for developing an actuarial liability model.

1. Various components are dependent on one another. Premium taxes are dependent on premiums, reserves are dependent on premiums and benefits, and premiums are

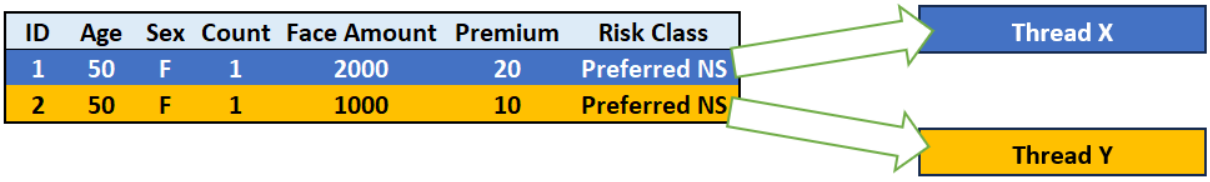
[3] This is often called a seriatim model which means cell-by-cell model.
[4] A decrement represents an event that causes a change to the status of the insurance policy (e.g., death).
[5] The formulas could be revised to include multiple decrements. Example: ${}_tP_x = {}_{t-1}P_x * (1 - {}_1w_{x+t-1}) * (1 - {}_1q_{x+t-1})$ where w corresponds to the annual lapse rate and q for mortality.
[6] See my previous post, Why We Set Reserves? This formula is a bit simplified, in that benefit and premium assumptions for calculating reserves will vary. Reserving method will vary by regulatory regime, by product and potentially by company.

dependent on decrements. These will be challenges since GPU calculations benefit from non-dependent tasks.

2. For this paper, I will assume the premium tax rate will be a constant 2%[7]. Therefore, under this assumption, the premium tax rate should be calculated once, at the end after all the premiums have been summed together at a specific time t.
3. There are likely multiple people aged x and of risk class r. Therefore, there is an opportunity to avoid performing calculations multiple times. This is a memory management comment. For instance, the model could group all insured in the same risk class into one grouping of threads, called a warp (Soyata, 2018).
4. Key to parallelization; The calculation of premiums, benefits, for insured i does not depend on insured i+1 (or any other insured). The dependences in this projection lie within the projection for each individual "i". Essentially calculations are time dependent and/or dependent on other values (see comment 1).

Based on the comments above, we should design our model to perform minimal number of calculations possible to maximize efficiency. Therefore, the order we perform calculations matters. As previously mentioned, calculating premium taxes (premiums * 2%) at the end allows us to perform this calculation only once. This is in contrast to performing it for each insurance policy. In addition, it would desirable to group similar insurance policies based on risk class, r (which can be based on gender, smoking status, or other allowable rating variables) to avoid performing duplicate calculations. For instance, in an actuarial liability model, two females aged 50 who do not smoke ("Preferred NS" in figure below), have the same projection values and it would be unnecessary to calculate them twice. Now, let us look at my proposed ordering which improves efficiency.

A. 1 Insured - 1 cell



B. Group Insureds into cells

My first step is grouping each insured into homogenous classes which was discussed in the previous section. The figure above highlights the efficiency achieved by grouping into cells. Scenario A represents calculating each insured separately. Scenario B realizes that this is inefficient. Thus, scenario B aggregates by face amount and premium and groups the data by risk

[7] Premium tax rate of 2% is a common assumption. In practice, premium tax rates vary by state (Grace, Sjoquist & Wheeler, 2007).

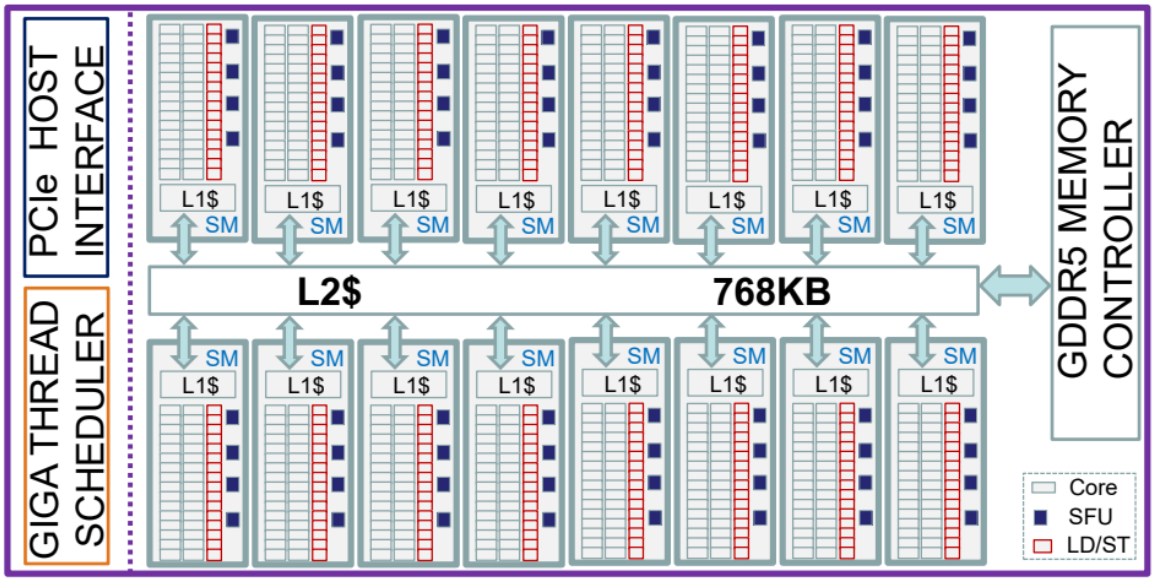
class, age and sex. This step may be done in the input file (outside of the model) or it can be done within the model using group-by querying logic. This grouping sets the model up to iterate through each cell rather than each individual insured. The next step is to develop survival curves and decrements. Survival curves represent the projected number of people who will keep their policies inforce. Based on these curves, we estimate the expected number of claims. Since each cell is independent of one another, this step will leverage parallel processing. After survival curves and decrements, the next step is to calculate insurance cash flows. The order of the cash flows should be specified or tiered. To repeat a previous example; since premium taxes are dependent on premiums, premium taxes should be calculated after premiums. While this is simplified model, my argument here is that the optimal actuarial model should have specified levels specifying the calculation order. The last step in this model is to calculate reserves. Reserves are calculated last, since they are calculated based on all future premiums and benefits and must be calculated after all premiums are developed.

Memory Management

Before going any further, I want to make one thing clear. GPUs are not a godsend as GPU-based parallel processing is not without tradeoffs. GPU-based parallel processing speeds up calculations by optimizing computer resources. GPUs can run thousands of calculations (threads) at the same time but can be limited by memory. Since there are thousands of threads the memory across all thread adds up very quickly (Soyata, 2018). CPU-based parallel processing is less susceptible to this problem. While GPUs can run out of memory fairly quickly, there are some mitigations. Before discussing mitigations, I will first discuss the memory structure of a GPU.

In a GPU there are several "streaming multiprocessors" (SM). Each SM is responsible for coordinating work among the cores inside the SM. With innovation, the number of SM's crammed into GPUs increased (Soyata, 2018). Each GPU has a pool of shared memory, called L2$ and all streaming multiprocessors share (Soyata, 2018). Within each SM, there are several cores. The cores are responsible for processing computational work. Within each SM, there is a second pool of shared memory that the cores share, called L1$ (Soyata, 2018). In addition, within the SM there are various load/store units whose job is to queue memory load and store requests (Soyata, 2018). To give some sense for numbers, the NVIDIA A100 GPU has L1$ memory of up to 192 KB per SM and L2$ memory of up to 40 MB (NVIDIA, 2020). It is important to note that as microchip engineering innovated, memory in GPUs increased over time. This theme is consistent with Moore's law that states computing power doubles every few years (Miller, 2022 & Soyata, 2018). This may make GPU-based actuarial modeling more feasible. Below is a picture from Soyata (2018) of the architecture for one of NVIDIA's GPUs, the "Fermi family" of GPUs. Note that later families were able to cram more components and memory, I have shown the Fermi family because this picture is easier to read[8].

[8] In picture below, LD/ST = Load/store units, SFU = special function units.



We will now move on to programming techniques. Below is a list of ways to improve processing times per Soyata (2018).

1. Limiting interaction with DRAM.
2. "Efficient calculations."
3. Implementing a cache.

The first way I will discuss is how to avoid accessing Dynamic Random Access Memory (DRAM) repeatedly. DRAM is great for storing vast reams of data. However, reading and writing to DRAM is slow (Soyata, 2018). Static Random Access Memory (SRAM) is much quicker but the drawback is the storage space is lower. As such when performing actuarial calculations, it may be useful to pull in data from DRAM (e.g., mortality rates) and store it to SRAM temporarily. This will save computing time if there are repeated lookups since the model reads data from DRAM less often. In addition, it can also be useful to apply this technique when writing data to permanent memory. When writing to permanent memory it is more efficient to write large chunks rather than small bits (Soyata, 2018).

The second area is "efficient calculations." This was labeled as such because certain operations take longer than others to calculate. For example, multiplication is faster than division (Soyata, 2018). An application of multiplication over division, could be present values.[9] When developing discount factors for present value calculations, it will likely be more efficient to input discount rates rather than effective interest rates[10] to avoid division. It is also helpful when iterating through scenarios, to avoid calculating variables more times than needed. For instance, let us say we are looping through time to project premiums. To calculate premiums, we need to calculate the premium rate for each insured ("i" from previous section). I contend that the premium rate

[9] Actuarial models will calculate many present values. E.g., Benefit Reserves = PV of Benefits – PV of Premiums.
[10] Let discount rate = d and effective interest rate = i. Discount Factor for time 1 = $v(1) = (1+i)^{-1} = (1-d)^{1}$ => $d = 1 - (1+i)^{-1} = i / (1+i)$. Discount rates and interest rates are often used interchangeably (I am guilty of this) but technically there is a difference. For this paper, I have used the technical definitions.

only needs to be calculated once at the beginning, not at each projection increment.[11] By calculating the premium rate at the start, you save n – 1 steps (n is the total number of time increments). Thus, moving certain calculations outside the inner loop speeds up runtime.

The last area I will discuss is utilizing the concept of a cache. This is less of a parallel processing suggestion but rather a general improvement for actuarial models. A cache saves recent data to memory for future use (Soyata, 2018). For instance, let us explore a scenario where an actuarial model calculates the liability projections first and then calculates assets cash flows thereafter (This is an extension of the actuarial liability model[12]). Hypothetically, let us say there is an error in the asset portion of the model. It would not make sense to recalculate the liability model. This mistake would become a time-consuming one as liability models can take hours to run. Therefore, the actuarial model would benefit from having the liability results cached. If the model needs to rerun, the model should skip over recalculating the liability projections and go straight to the asset portion. A drawback with the cache is it will eat into memory, and it may not be feasible to store all projections on a granular basis. However, more aggregated calculations such as assets and capital (especially capital) may be useful applications of the cache which do not depend on granular liability projections.

Now, all these techniques may not be sufficient to save GPU-based actuarial models from computer overload because actuarial models are rather complex. The actuarial liability model may need to be broken up into separate calculation where each piece is run using parallel processing and after that piece is run, it is stored into DRAM. The reading and writing to and from DRAM will raise run time but may be needed if the GPU memory per thread is a constraint. Further, I am by no means an expert in computer science, parallel processing or GPUs but, I do see an opportunity to improve actuarial models. This same technology has allowed artificial intelligence to make giant leaps by reducing the time needed to train large AI language models. I don't believe that actuarial calculations are inherently more complicated than the matrix math used in AI models and believe that improvements in computing could save the profession time.[13]

[11] Assume that premium rates are fixed in this example. Even if they were not fixed you should still calculate the premium at the start but calculate a vector of premiums.
[12] Depending on the modeling software, assets and liabilities can be modeled together or separately.
[13] Hopefully AI doesn't take actuaries' jobs!

References

Grace, M., Sjoquist, D. L., & Wheeler, L. (2007). Insurance premium taxes. In *Proceedings. Annual Conference on Taxation and Minutes of the Annual Meeting of the National Tax Association* (Vol. 100, pp. 34-42). National Tax Association.

Harding, S. (2022). What is a CPU core? A basic definition. Tom's Hardware. https://www.tomshardware.com/news/cpu-core-definition,37658.html

Miller, C. (2023). *Chip War*. Simon & Schuster UK.

NVIDIA. (2020). Nvidia A100 tensor core GPU architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Richter, F. (2023). Infographic: Big three dominate the Global Cloud Market. Statista Daily Data. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

Soyata, T. (2018). GPU parallel program development using CUDA. CRC Press, Taylor & Francis Group.