Predictive Analytics and Futurism
December 2016
Issue 14

Introduction to Using Graphical Processing Units for Variable Annuity Guarantee Modeling

By Bryon Robidoux

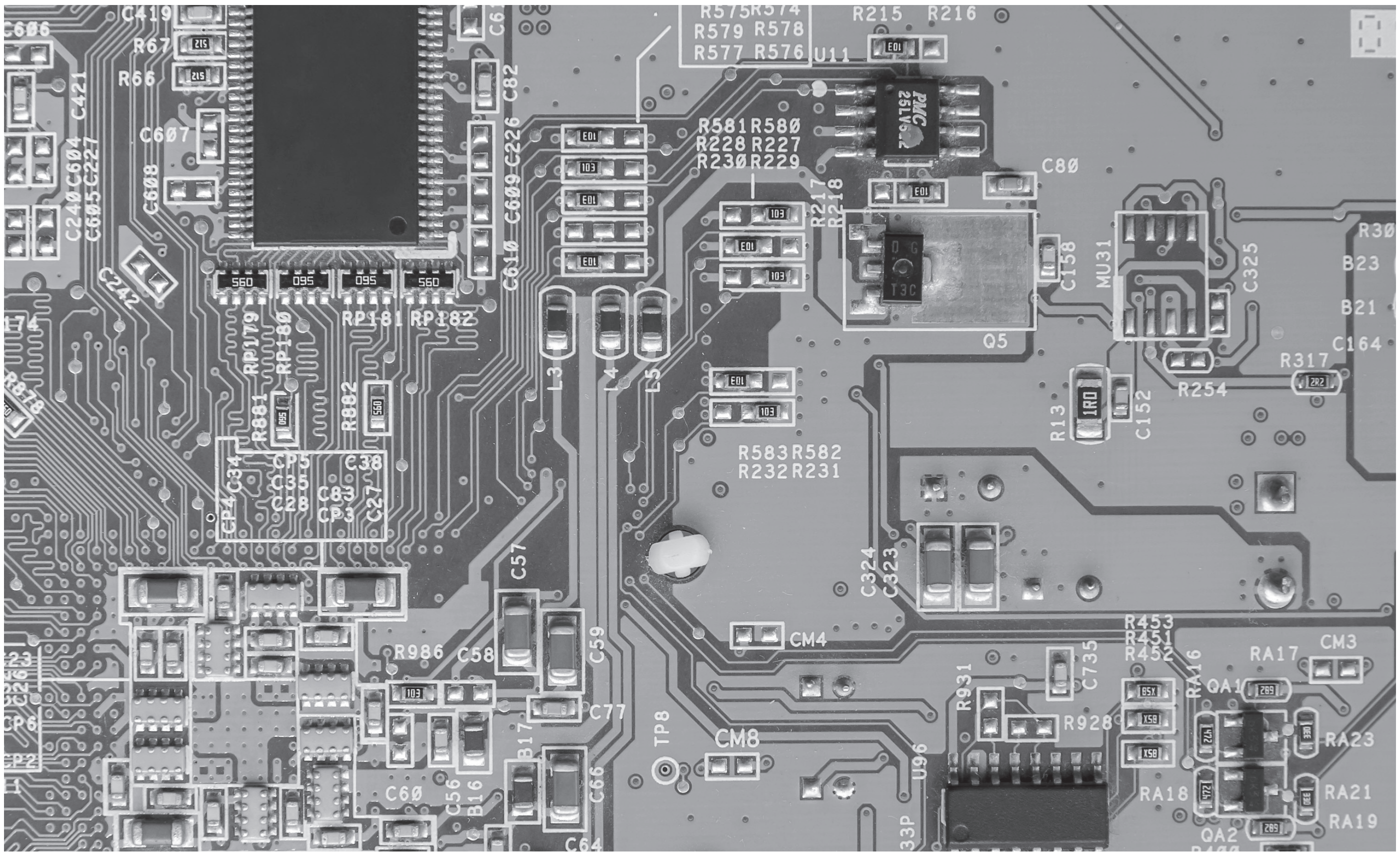
Recently, I was asked to give a webcast on using Graphical Processing Unit (GPU) for predictive modeling. This paper will be an introduction to the GPU and will be a precursor for the webcast. A GPU is nothing more than the graphics card in your computer which creates the images on your monitor. The laptop I am working on now has four cores in its Central Processing Unit (CPU) where a GPU will have thousands of cores. Each core allows calculations to be done in parallel. It became apparent to scientists that a computer's GPU was a cheap way to get massive parallelization on a desktop computer. NVIDIA, a company that manufactures GPU cards for computers, introduced CUDA (originally, an acronym for Compute Unified Device Architecture) extensions to the C programming language and CUDA cores to encourage scientists to use GPUs for scientific computing. Originally scientists had to transform their calculations to fool the GPU, but now most NVIDIA graphics cards are CUDA compliant. This manor of using a GPU is quickly going from cutting-edge to mainstream in many different scientific professions, but I don't think this is true yet for the actuarial profession.

As I have been approached about working on various GPU projects, the person touting the project usually describes it as fairly straightforward. They state that all that is required is to take the existing model, change a few lines of code, and abracadabra the new calculations will run just as fast as a compute cluster. This could not be further from the truth! This mentality may get something to work, but it will be nowhere near the speed advertised or possible. It may actually be slower than no GPUs at all! Building a variable annuity guarantee model is way different than building a predictive model, but at the same time, you will potentially run into similar types of bottlenecks and issues. Even though 99 percent of the time you will be using a library such as Theano, which is a high-level python library, or Thrust, which is a C++ library, to do the predictive modeling on GPUs, understanding the finer points of GPU architecture will help you understand why a particular model is running slower than anticipated or why it may not be able to be ported to a GPU. The first part of this article will give a high level overview of the GPU architecture. The second part of the article will describe different aspects of modeling a variable annuity guarantee. The third part of the article will try to combine the constraints of the GPU architecture with the common features of the variable annuity guarantee model to create solutions to likely bottlenecks in the variable annuity guarantee model.

In GPU literature, the CPU is called the host, whereas the GPU is called the device. Other than the host calling functions to execute on the device, the host and device run separately from each other. The host and device have their own memory also. To use the device, the data must be migrated from the host memory into the device's global memory. This is one of the first barriers to using a GPU. It can take longer to transfer data to and from

FEATURE*	TESLA K80	FEATURE	KEPLER GK210
GPU Chip(s)	2x Kepler GK210	Compute Capability	3.7
Peak Single Precision (base clocks)	5.60 TFLOPS (both GPUs combined)	Threads per Warp	32
Peak Double Precision (base clocks)	1.87 TFLOPS (both GPUs combined)	Max Warps per SM	64
Peak Single Precision (GPU Boost)	8.73 TFLOPS (both GPUs combined)	Max Threads per SM	2048
Peak Double Precision (GPU Boost)	2.91 TFLOPS (both GPUs combined)	Max Thread Blocks per SM	16
Onboard GDDR5 Memory1	24GB (12GB per GPU)	32-bit Registers per SM	128K
Memory Bandwidth1	480 GB/s (240 GB/s per GPU)	Max Registers per Thread Block	64K
Achievable PCI-E transfer bandwidth	12 GB/s	Max Registers per Thread	255
# of SMX Units	15	Max Threads per Thread Block	1024
# of CUDA Cores	2880	Shared Memory Configurations	16KB + 112KB L1 Cache
Memory Clock	3004 MHz	Max Shared Memory per Thread Block	48KB
GPU Base Clock	745 MHz		

*table from https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-kepler-gpu-accelerators/



the host and device than it does to run the actual calculation. In these types of situations, it is best not to use the device. The device contains one or more kernels. Kernels are major functions used to run code on the device. It consists of many blocks that work independently of each other. Each block consists of a user defined number of threads. The threads are what actually execute the code.

The key to understanding the bottlenecks when developing for GPUs is to understand how the threads get scheduled and the memory resources available to the threads. There are multiple layers of schedulers. The top level is GigaThread global scheduler which controls the scheduling of the kernels and the streaming multiprocessors (SM). Within the global scheduler, the SMs control the scheduling of the blocks. Each SM schedules its blocks independently of the other SMs. This is why if synchronization is required among threads, it must happen within the block. The SM's basic unit of scheduling threads is the warp, which is a block of 32 threads. The compute capacity is defined by NVIDIA as the maximum number of resident threads per SM.1 The larger the compute capacity the better the GPU is suited for scientific calculations.

For this article I am going to use the Kepler K80 GPU for the example. This is worthy of doing actuarial modeling. Don't be fooled into thinking that your gaming graphics card is good for actuarial modeling. Even though it probably contains CUDA cores, the compute capacity is not sufficient. It would be worthy for proof of concept, but not production requirements. The following table shows the specifications for the K80.

A GPU is a Single Instruction Multiple Data (SIMD) device. This means that all the threads within a warp must process the same instruction in order to run in parallel and only the data can be different. Warp divergence occurs if all the threads in the warp are not running the same instruction. This means that an "if statement" can have a huge impact on speed because this is a natural place for instructions to diverge. In a worst case scenario, each thread within the warp will have to be run serially because each thread has to run a separate instruction, in the case of the K80, causing a potential 32X slowdown to occur.1 The next large hurdle is memory resources.

The following table gives the specifications on different types of memory within the GPU. The important information to get from the table is where the memory is located and its bandwidth. For this article, the local memory will be restricted to just L1-Cache. (A level 1 cache (L1 cache) is a memory cache that is directly built into the microprocessor, which is used for storing the microprocessor's recently accessed information, thus it is

MEMORY	LOCATION	CACHED	ACCESS	SCOPE	BANDWIDTH GB/S*	ON-CHIP/OFF-CHIP
Register	On-Chip	No	Read/write	One Thread	10,847	45
Local	On-Chip	Yes	Read/write	One Thread	2,169	9
Shared	On-Chip	N/A	Read/write	All threads in the block	2,169	9
Global	Off-Chip (unless cached)	Yes	Read/write	All threads + host	240	1
Constant	Off-Chip (unless cached)	Yes	Read	All threads + host	240	1

* This table is from 1. It only provides the information for the Fermi C2050 on page 101. The values for shared, local and register are derived by multiplying the ratio of the memory bandwidth between the K80 and Fermi C2050 which is 240/177 to keep the relative speeds the same between the GPU models.

also called the primary cache.2) The on-chip memory is inside the SM, whereas off-chip memory sits outside of the SM. It is easy to see that on-chip memory is faster than off-chip memory. The problem is that the faster the memory, the less of it there is available.

From the K80 specification table, the global memory size is 24GB whereas the register memory size is 128KB per SM. Register memory is a very valuable and limited resource. The only memory fast enough to keep the GPU running at full capacity is the register memory. The global memory is nowhere near fast enough. For example, a very simple C++ code snippet would be

for(i=0;i<N;++i) c[i]=a[i]+b[i];

where a, b and c are all in global memory and single precision 4-byte variables. This simple little program requires two memory reads from a and b and one memory write to c. In order to keep the 5.6 trillion floating point operations (TFlop) busy on the K80, there would need to be 3 instructions * 4 bytes * 5.6 Tflops = 67.2 Terabytes/second (TB/s) of memory bandwidth, but there is currently only 240 GB/s available.1 It is obvious that a major factor of making the device run faster is developing a good strategy for moving data from the global memory to the register memory. It is very easy to create a situation called register spillover which occurs when a thread block requires more registers than are available. In early generations of the device, the register spillover went into global memory, but for later generations the spillover first goes to the L1-Cache and if that is exhausted it spills over into global memory. The L1-Cache is also a limited resource and needs to be managed carefully, but it does reduce some of the penalty of register spillover. The problem with the slow memory is that the warp will not schedule threads to run until all its resources are available. This means that very few warps can operate in parallel because the required data is in a traffic jam. The key strategy in GPU programming is to maximize data reuse, so you avoid unnecessary trips to fetch data from global memory. Now that we know some of the major constraints of the GPU it is time to move onto the modeling of the variable annuity.

It is obvious that a major factor of making the device run faster is developing a good strategy for moving data from the global memory to the register memory.

This article will stay focused on the Guaranteed Minimum Withdrawal Benefit for Life (GMWBL) rider. The best way to approach it is to break down the rider into its fundamental modeling components so we can best try to map them to the GPU. From the top down, there is obviously the rider plan, policy, withdrawal cohorts for the policy, the time steps and lastly the order of transactions within the time step. The withdrawal cohort is just specifying the likely time someone will exercise their benefit along with the probability of them doing it at that time. From all the models I have worked on, they follow Mary Hardy's suggestion to make the time of withdrawal deterministic.3 At the very least, withdrawal cohorts are dimensioned by issue age, with four to 10 per issue age, but I have also seen them dimensioned by qualified and non-qualified status. Qualified status means the policy was purchased with proceeds from a before-tax account such as a 401k. Qualified policies will have significantly different behavior from non-qualified policies. Depending on the time of withdrawal, the policyholder can be rewarded through a credit or penalized through a loss of benefit. The order of transactions is my way of generalizing Mary Hardy's3 characterization of her two transaction types, which were before fees and after fees. In practice, there are usually three or four transaction types. They are usually label beginning of period (BOP), middle of period (MOP), and end of period (EOP). BOP is when the market mechanics, such as fund returns, and withdrawal behavior are calculated. MOP is when fees are applied and any ratchets or rollups occur, if applicable. EOP is when the decrements are applied such as mortality and dynamic lapse. Now that the fundamental components of modeling have been established, it is time to combine the GPU and the variable annuity guarantee model to see where potential calculation bottlenecks and problems can occur.

From an actuarial perspective, the most intuitive way to model a GMWBL rider would be to put each withdrawal cohort of a policy on its own thread and to sort the policy file by rider plan and policy number. (Modeling at any lower level doesn't make sense because by definition they are serially dependent.) A few simple examples will show that this will easily lead to a massive register spillover and warp divergence if done haphazardly. Currently, I am working on a GMWBL model that requires 423 bytes of inputs to model a single withdrawal cohort of a policy. The inputs are a mixture of Booleans, single precision floating-point numbers, and integers. If each withdrawal cohort of a policy is allocated to a thread this implies 2048 threads *423 bytes = 866KB of register memory is needed to calculate the block. There is only 128KB of registers available for the block so the capacity of the registers has been exceeded by 6X. Even with the ability to spill over into the L1-cache, the L1-cache is 112KB so this is still insufficient to handle all the data. If I were to port my current model to a GPU, as is, there is little I could do to avoid register spillover and not have huge speed reduction. The purposed strategy will also lead to warp divergence because the consequences and rewards to the policyholder depend on the timing of withdrawal. The consequences and rewards cause each withdrawal cohort to have a different behavior, which leads each withdrawal cohort down a different logical path.

In order to address the issues above, the minimum requirement is to perform some preprocessing. In order to get the data size requirement down, there needs to be a strategy for data reuse. Some of this can be accomplished by strategically sorting the policy file and creating a sort key by rider plan, then by issue age, then by assumed withdrawal time, then by qualified status, and then by any other fields that cause material changes to calculation behavior. This forces the policies with the most homogenous information and behavior together. Within the code, I would force all blocks to be homogenous by requiring only policies with the same sort key to be in a block. Enforcing homogeneity through the sort order and code should help to reduce thread divergence. It should also reduce register spillover and promote data reuse because roughly a quarter to a half of the 423 bytes of the data for the GMWBL policy are to describe variations within the rider features which are not policy specific. The common rider features can be migrated to shared memory and shared among threads. The shared memory has the speed as L1-cache, which is much better than global memory. The idea of grouping homogenous policies together to speed up calculations should not be unfamiliar to modeling and valuation actuaries because this is very similar exercise to get good cell compression on policy files.

One last topic that should be mentioned is the creation of the market dynamics from the economic scenario generator (ESG). The gold standard of random number generators (RNG) for use in ESG is the Mersenne Twister, because of its enormous periodicity. The Mersenne Twister is a serial generator because, after the seed is applied, each number generated depends on the previous number generated. It may be tempting to think that each thread should receive its own seed, but this would likely not preserve the statistical properties of the random number generator such as mean and standard deviation. In order to work properly, it is highly recommended that cuRand® be used to generate random numbers. The RNGs have been specifically designed, such as the MTGP Mersenne Twister, so that each thread can generate its own set of random numbers and still preserve the proper statistical properties. At this point, the only issue is with testing. It is very likely that individual policies would be checked with Excel which implies the testing will use the original Mersenne Twister. The original Mersenne Twister and the MTGP Mersenne Twister will not produce the same set of random numbers. They are only guaranteed to have the same statistical properties. As a part of testing, it will be required to isolate the random numbers from the device so calculations will match.

In conclusion, modeling variable annuities on GPUs can be a fun and challenging problem. It is not a simple migration to rework a model built for a compute grid to work on a GPU. This article by no means addresses all the issues of modeling VA riders with a GPU, but it should give you a good flavor of the types of issues that can occur. Even though building a variable annuity model on a GPU is much different than building a predictive model with a library such as Theano or Thrust, it should give you a good appreciation for some of the challenges of creating those libraries, demonstrate some underlying reasons on why the model may be calculating slower than expected, and possible ideas on how to speed the model up. ■

Bryon Robidoux, FSA, is director and actuary, at AIG in Chesterfield, Mo. He can be reached at *Bryon.Robidoux@aig.com*

ENDNOTES

1 Rob Farber, CUDA Application Design and Development, Morgan Kauffman 2011

2 *https://www.techopedia.com/definition/8048/level-1-cache-l1-cache*

3 Mary Hardy, Investment Guarantees: Modeling and Risk Management for Equity-Linked Life Insurance.

4 *https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-kepler-gpu-accelerators/*