Tensor Processing Units for Financial Monte Carlo

Francois Belletti, Davis King, Kun Yang, Roland Nelet, Yusef Shafi, Yi-Fan Chen, John Anderson

Google Research
Mountain View CA
USA
belletti@google.com

***Abstract*—Monte Carlo methods are core to many routines in quantitative finance such as derivatives pricing, hedging and risk metrics. Unfortunately, Monte Carlo methods are very computationally expensive when it comes to running simulations in high-dimensional state spaces where they are still a method of choice in the financial industry. Recently, Tensor Processing Units (TPUs) have provided considerable speedups and decreased the cost of running Stochastic Gradient Descent (SGD) in Deep Learning. After having highlighted computational similarities between training neural networks with SGD and stochastic process simulation, we ask in the present paper whether TPUs are accurate, fast and simple enough to use for financial Monte Carlo. Through a theoretical reminder of the key properties of such methods and thorough empirical experiments we examine the fitness of TPUs for option pricing, hedging and risk metrics computation. We show in the following that Tensor Processing Units (TPUs) in the cloud help accelerate Monte Carlo routines compared to Graphics Processing Units (GPUs) which in turn decreases the cost associated with running such simulations while leveraging the flexibility of the cloud. In particular we demonstrate that, in spite of the use of mixed precision, TPUs still provide accurate estimators which are fast to compute. We also show that the Tensorflow programming model for TPUs is elegant, expressive and simplifies automated differentiation.**

***Index Terms*—Financial Monte Carlo, Simulation, Tensor Processing Unit, Hardware Accelerators, TPU, GPU**

I. INTRODUCTION

The machine learning community has developed several technologies for speeding up Stochastic Gradient Descent algorithms for Deep Learning [18], including new programming paradigms, special-purpose hardware, and linear-algebra computation frameworks. This paper demonstrates that we can apply the same techniques to accelerate Monte Carlo integration of stochastic processes for financial applications.

A. Monte Carlo estimation in finance and insurance

A key problem when pricing a financial instrument — for insurance or speculation — is to estimate an average outcome defined by a probability space $(\Omega, \mathcal{F}, \mathbb{P})$:

$$E_{\mathbb{P}}\left[f(\omega)\right]$$

where E denotes the expectation. In the following we first provide a basic introduction to derivatives pricing and demonstrate how expectations lie at the core of pricing, hedging and risk assessment. We then introduce how Monte-Carlo methods are generally used to estimate such expectations and focus on the case in which the random fluctuations are generated by stochastic processes. After having introduced state-of-the-art methods to improve the statistical properties of such estimators, we show how hardware accelerators enable such estimators to be computed faster thanks to parallelization.

1) Stochastic processes in continuous time: Stochastic processes remain the main abstraction employed to model financial asset prices. Let us briefly introduce these theoretical constructs before we describe how they are practically approximated by numerical representations. Consider a filtered probability space $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ (where $\mathbb{F} = \{\mathcal{F}_t\}$ is the corresponding canonical filtration) supporting a q dimensional Brownian motion W and the Stochastic Differential Equation (SDE)

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t,\ t \in [0, T]\,. \tag{1}$$

The drift (μ) and volatility (σ) functions take values respectively in $\mathbb{R}^p$ and $\mathbb{R}^{p,q}$. By definition, a strong solution (X) to Eq. (1) is a process taking values in $\mathbb{R}^p$ such that $\int_{s=0}^{T} \left(||b(s, X_s)||_2 + ||\sigma(s, X_s)||_2^2\right) ds$ is almost surely finite and

$$X_t = X_0 + \int_0^t b\left(s, X_s\right) ds + \int_0^t \sigma\left(s, X_s\right) dW_s,\ t \in [0, T]\,.$$

Assuming that X_0 is a random variable with finite variance independent of W, and that $||b(\cdot, 0)||_2$ and $||\sigma(\cdot, 0)||_2$ are square integrable (as functions of t), then the existence of a finite Lipschitz constant K such that

$$||b(t,x) - b(t,y)||_2 + ||\sigma(t,x) - \sigma(t,y)||_2 \leq K||x - y||_2$$

for all $x, y \in \mathbb{R}^p$ and $t \in [0, T]$ guarantees the existence of such a strong solution on $[0, T]$.

2) Monte Carlo methods in finance and insurance: Monte Carlo methods rely on simulation and numerical integration to estimate $E_{\mathbb{P}}[f(X_T)]$ or $E_{\mathbb{Q}}[f(X_T)]$ under the historical or risk-neutral probability ($\mathbb{P}$ and $\mathbb{Q}$ respectively) [16]. Some contracts defining financial derivatives may specify a *path dependent* outcome—such as Barrier or Asian options—in which case the theory of Black, Scholes and Merton still leads us to estimate $E_{\mathbb{P}}[f(X_{0:T})]$ or $E_{\mathbb{Q}}[f(X_{0:T})]$ where $X_{0:T}$ denotes the observation of the process X on the interval $[0, T]$. In general, we therefore seek an estimator for an expectation of the type

$$E[f(X_{0:T})] \text{ where } (X_t) \text{ solves (1) on } [0, T]. \tag{2}$$

Monte-Carlo methods rely on numerical discretization and integration to produce an estimate for (2) in the form of an empirical mean over simulated trajectories $\left\{\widetilde{X}^n_{0,T} | i = 1 \dots N\right\}$:

$$\widehat{I}_N = \frac{1}{N}\sum_{n=1}^{N} f\left(\widetilde{X}^n_{0,T}\right). \tag{3}$$

In general, because the dynamics of (X_t) are specified in continuous time with real values, a computer-based simulation will suffer from bias coming from the limited precision in numerical representations and more importantly the temporal discretization. The variance of the the estimator (3) is also a problem: typically if it costs $O(N)$ Monte Carlo samples to produce a result with a confidence interval of size 1 then reducing the interval's size to ϵ comes at a cost of $O(\frac{N}{\epsilon^2})$ simulations. Furthermore, the compute time cost generally scales as $O(q^2)$ when correlations between different components of (X_t) are taken into account. For these reasons, Monte Carlo methods constitute some of the most computationally intensive tasks run routinely at scale across the financial industry. Accelerating Monte Carlo estimation and making it more cost effective has been a long standing challenge with huge repercussions for derivative pricing, hedging and risk assessment.

3) Greeks and sensitivity analysis: Monte Carlo methods in quantitative finance are also used to estimate sensitivities of derivatives' prices with respect to model parameters and the current state of the market. Sensitivities to market parameters—the financial "Greeks" [16]—are used not only to quantify risk, but also to construct hedges and synthetic replicating portfolios. For example, the "delta", the sensitivity of the price of an option with respect to the current price of the underlying(s), specifies the amount of the underlying(s) that needs to be held in a replicating portfolio. Automated differentiation of Monte Carlo pricers is now a solution of choice in quantitative finance as it is more computationally efficient than methods such as bumping to compute sensitivities with respect to many different inputs and parameters [31]. Tensorflow was designed with automated differentiation at its very core as this technique—often referred to as Back-Propagation—is of key importance when training Machine Learning by Stochastic Gradient descent [2]. Moreover, Tensorflow readily offers the opportunity to accelerate the forward simulation and automated differentiation by Back-Propagation without needing any additional code— all while enabling researchers to leverage modern hardware such as GPUs and TPUs for acceleration.

B. Contributions

In the present paper we focus on leveraging Tensor Processing Units (TPUs) for financial Monte Carlo methods. We aim to show that although such accelerators were designed primarily to accelerate the training of Deep Learning models by Stochastic Gradient Descent, TPUs provide cutting edge performance for Monte Carlo methods involving discretized multi-variate stochastic processes. In particular, we present the following contributions:

- We demonstrate that in spite of the limited numerical precision employed natively in matrix multiplications on TPU, accurate estimates can be obtained in a variety of applications of Monte Carlo methods that are sensitive to numerical precision.
- We argue that TPUs offer efficient risk assessment solutions: for risk metrics being able to simulate many more scenarios is of key importance, while in cases where limited precision could become an issue, Multi-Level Monte Carlo methods can potentially efficiently correct precision-related bias.
- We benchmark the speed of TPUs and compare them to GPUs which constitute the main source of acceleration for general purpose Monte-Carlo methods outside of Field Programmable Gate Arrays (FPGA) and Application-specific Integrated Circuit (ASIC) based solutions.
- We demonstrate that Tensorflow [2] constitutes a high level, flexible and simple interface that can be used to leverage the computational power of TPUs, while also providing integrated automated differentiation.

The present paper demonstrates that Tensorflow constitutes a flexible programming API which enables the implementation of different simulation routines while providing substantial benefits by running enabling the running of underlying computations in the Cloud on TPUs. A key consequence is that experiences that used to be iterative for developers now become interactive and inherently scalable without requiring investing in any hardware. We believe such improvements can make financial risk management more cost-effective, flexible and reactive.

II. Related Work

A. Pricing techniques and typical computational workloads

Understanding typical computational patterns underlying financial Monte Carlo simulations is a first step towards acceleration. We now examine three typical computational workloads corresponding to different types of simulations routinely employed to price derivatives and assess risk in quantitative finance.

1) SIMD element-wise scalar ops in Euler-Maruyama discretization schemes: A first characteristic computational workload is associated with mono-variate geometric Brownian models and their extensions in the form of local [12], [19] or stochastic volatility models [5], [13]. The Euler-Maruyama scheme discretizes SDE (1) explicitly forward in time. Consider the simulation of N independent trajectories,

$$\widetilde{X}^n_{t_{i+1}} = \widetilde{X}^n_{t_i} + \mu\left(t_i, \widetilde{X}^n_{t_i}\right)\Delta_{t_i} + \sigma\left(t_i, \widetilde{X}^n_{t_i}\right)\sqrt{\Delta_{t_i}} Z^n_{i+1} \tag{4}$$

for $n = 1, \dots, N$ where $\widetilde{X}^n_{t_0} = X_0 \in \mathbb{R}$, $\Delta_{t_i} = t_{i+1} - t_i$, Z^n_{i+1} are Pseudo-Random Numbers distributed following $\mathcal{N}(0,1)$. In the uni-variate case, where the process being simulated has a single scalar component, implementing Equation (4) reduces to scalar add/multiplies which are independent across simulated scenarios. Simulating a batch of scenarios under this discretized scheme is therefore embarrassingly parallel and a clear example of a Single Instruction Multiple Data (SIMD) setting where the different elements of the data undergo independent computations. Such simulations are trivial to parallelize along the batch of simulated independent scenarios provided Pseudo-Random Numbers can be generated in parallel and in a consistent manner [9], [25], [30], [32].

2) Matrix-multiply ops in Multi-variate simulations of correlated processes: The Euler-Maruyama discretization scheme for scalar stochastic processes naturally extends to the multivariate setting where each stochastic process takes values in $\widetilde{X}_{t_i}^n \in \mathbb{R}^p$. Computationally, a major difference arises however. If the underlying Brownian motion is in $\mathbb{R}^q$, each simulated time-step in each scenario will require calculating $\sqrt{t_{i+1}-t_i}\sigma\left(t_i, \widetilde{X}_{t_i}^n\right) Z_{i+1}^n$ with $Z_{i+1}^n \sim \mathcal{N}(0, I_q)$ and $\sigma\left(t_i, \widetilde{X}_{t_i}^n\right) \in \mathbb{R}^{p,q}$, which implies that a $p \times q$ matrix/vector product has to be computed. If N scenarios are stacked together to benefit from the corresponding hardware acceleration, the operation becomes a $p \times q, q \times N$ matrix/matrix products.

3) Chained linear system inversions in the Longstaff-Schwartz Method (LSM) for value estimation: The regression-based estimation method proposed by Longstaff and Schwartz [24] to price American Options has become a standard pricing method for callable financial instruments (e.g., American or Bermuda options) with high dimensional underlyings (e.g., call on a maximum or weighted combination of stocks). In the setting of callable options, which can be exercised at multiple points in time before their maturity, the pricing problem is solved using a Monte-Carlo method to simulate future trajectories as well as a dynamic programming approach to back-track optimal decisions in time. To enable dynamic programming, for each decision instant t_i, one has to estimate a Value Function on the state of the underlying:

$$X_{t_i} \mapsto V_{t_i}(X_{t_i}) = \max\left(f(X_{t_i}), E(V_{t_{i+1}}|X_{t_i})\right)$$

with the convention that $V_T = f(X_T)$ where f is the option's payoff function. By definition, the conditional expectation $E(V_{t_{i+1}}|X_{t_i})$ is the closest square integrable random variable (according to the L_2 norm) to X_{t_i} . Therefore the LSM fits a model to interpolate $E(V_{t_{i+1}}|X_{t_i})$ between values of X_{t_i} that have actually been simulated. LSM employs a linear-regression on a set of K features derived from the the simulated values of X_{t_i} such as $(1, X_{t_i}, X_{t_i}^2, \dots)$ or a finite number of Hermite polynomials evaluated at the simulated values [16], [24]. Given a set of simulated values $\left\{\widetilde{X}_{t_i}^n | n = 1 \dots N\right\}$, the set of values $\left\{V\left(\widetilde{X}_{t_i}^n\right) | n = 1 \dots N\right\}$ is projected onto the set of regressors $\left\{\psi_1\left(\widetilde{X}_{t_i}^n\right), \dots, \psi_K\left(\widetilde{X}_{t_i}^n\right), | n = 1 \dots N\right\}$ where $(\psi_k)_{k=1 \dots K}$ are featurizing functions (e.g., Hermite polynomials). Therefore, a linear regression of N scalar observations is needed, for each candidate time-step to exercise the option, onto N vectors of K dimensions. Typically, for efficiency, a Cholesky decomposition of the Grammian will be computed prior to effectively solving the linear regression. This computational cost adds to the cost of simulating the original paths for the trajectory of the underlying asset(s) which may themselves be correlated. The overall procedure yields a price estimate as the expected value function at the first exercise time:

$$\widehat{I}_N = \frac{1}{N} \sum_{n=1}^{N} V\left(\widetilde{X}_{t_0}^n\right). \qquad (5)$$

B. Bias reduction for reduced precision Monte Carlo

Multiple techniques have been developed to improve the convergence properties of the above estimators in order to spend less computational power for a given amount of variance (and bias). The Application Programming Interface (API) exposed by Tensorflow (see Figure 10 for a code snippet) readily enables variance mitigation schemes such as antithetic sampling, control variates and importance sampling.

1) Control and antithetic variates: The most widely known strategy to decrease the variance of a Monte-Carlo method employing Pseudo-Random Numbers (PRN) generated according to a symmetrical probability distribution is perhaps the antithetic variate approach [16], [28]. Applied to Equation (4), the method works as follows: for each trajectory relying on a series of generated PRN $(Z_i^n)_{i=1,\dots N}$ (distributed following $\mathcal{N}(0,1)$), a symmetrical trajectory using the sequence opposite numbers as Gaussian PRNs is employed. As the outcomes corresponding to these symmetrical trajectories are averaged out in the final Monte-Carlo estimate, variance is reduced without introducing bias.

2) Importance Sampling: Importance Sampling effectively reduces the variance of Monte-Carlo estimators concerned with rare events which are useful when pricing barrier options with very high or low activating barriers or for Value-at-Risk (VaR) estimation. As VaR requires the estimation of the $95^{\text{th}}/98^{\text{th}}/99.9^{\text{th}}$, it can be worthwhile encouraging the simulation to simulate for extreme outcomes which [17] enabled via Importance Sampling. Importance Sampling weights are derived analytically and prevent the simulation under a probability distribution different from $\mathbb{P}$ or $\mathbb{Q}$ from introducing bias. Recent developments [4], [23] enable the automation of the search for a proposal distribution for the simulations that mitigates variance optimally. Stratified sampling has also proven effective in accelerating Monte Carlo convergence [28].

3) Multi-Level Monte-Carlo: Multi-Level Monte Carlo (MLMC) constitutes an exciting development to accelerate Monte-Carlo methods thanks to mixed precision computation schemes [14], [15]. We present the method in the two-level setting. Let us assume that two different programs are available to compute a trajectory given a sequence of PRNs $(w_i^n)_{i=1 \dots I}$: an expensive high definition program P_H and a relatively inexpensive low definition program P_L. While the standard Monte-Carlo method would employ $\frac{1}{N}\sum_{n=1}^{N} P_H\left((w_i^n)\right)$ to estimate the expected outcome, the MLMC method instead employs

$$\frac{\sum_{n=1}^{N_H} P_H\left(\left(w_i^n\right)\right)-P_L\left(\left(w_i^n\right)\right)}{N_H}+\frac{\sum_{n=N_L}^{N_L+N_H} P_L\left(\left(w_i^n\right)\right)}{N_L} \quad (6)$$

typically with $N_H << N_L$. Indeed, as the same perturbations are injected in P_H and P_L, generally $\operatorname{Var}\left(P_H\left(\left(w_i^n\right)\right)-P_L\left(\left(w_i^n\right)\right)\right) << \operatorname{Var}\left(P_L\left(\left(w_i^n\right)\right)\right)$. A way to interpret MLMC is that the first N_H PRN sequences are spent in MLMC to estimate the bias induced by the use of P_L instead of P_H. Only a few samples are needed because the estimator's variance is small as consistent PRN sequences are injected into P_L and P_H. The subsequent N_L PRN sequences are used in a large number to reduce the variance of the average-based estimator with the less expensive P_L. A typical example of MLMC consists in considering the same simulation program running in double precision for P_H and half precision for P_L. Beyond simply lowering the precision of the floating point representation, P_L can be a radically coarser counterpart to P_L as in [28] with a coarser temporal discretization or a coarser spatial discretization through quantized states.

C. Pre-existing hardware acceleration strategies

Having reviewed strategies relying on algorithmic modifications to accelerate the convergence of Monte-Carlo estimators, we now give an overview of hardware-based techniques to reduce their running time. A first approach to accelerate Monte-Carlo methods had consisted in running them on High Performance Computing (HPC) CPU grids with parallelization paradigms such as Message Passing Interface (MPI). We focus here on device-level acceleration with hardware accelerators that can be used as elements of a distributed compute grid if needed.

1) GPUs: The rise of general purpose high level APIs to orchestrate scientific calculations on GPUs with CUDA or OpenCL has prompted a wide development of GPU-based approaches to accelerate Monte Carlo methods. Pricing and estimating risk metrics in finance are especially well-suited to acceleration by GPUs, due to the embarrassingly parallel nature of Monte Carlo Methods, and to their use of computationally intensive linear algebra routines. Methods enabling the generation of PRNs in parallel correctly and efficiently [9], [25], [30], [32] coupled with algorithmic re-factorization to fully utilize GPUs have enabled substantial speedups with respect to CPUs for pricing [3], [22], [26], [29], risk metrics [10] and sensitivity analysis [11].

2) FPGAs: Field Programmable Gate Arrays (FPGAs) have grown popular to accelerate Monte-Carlo methods and represent more specialized and energy efficient competitors to GPUs. Many works have demonstrated that FPGAs provide substantial speedups with respect to GPU implementations and reduce energy costs in servers. While some methods have employed FPGAs as standalone solutions [33], [34] other approaches have used a mixed precision approach relying on both a CPU and an FPGA [6], [7]. In particular, MLMC [15] can be applied to FPGAs computing low resolution fast simulations paired with CPUs running an implementation at reference precision.

III. TENSOR PROCESSING UNITS

A Tensor Processing Unit ("Cloud TPU" or "TPU" for short)—a custom-developed application-specific integrated circuit (ASIC) specialized for deep neural networks—offers 420×10^{12} floating-point operations per second (FLOPS) and 128GB of high bandwidth memory (HBM) in its latest release. The TPU architecture is abstracted behind the Tensorflow framework. High-level Tensorflow programs, written without using detailed knowledge of TPUs, both for for training large deep neural networks and for performing low-latency online prediction can be deployed on TPU hardware in the cloud. [20] reports impressive acceleration of training and online prediction.

Although TPU targets deep learning, it is designed for maximum performance and flexibility to address computational challenges in various fields. In the present paper, we particularize its application to financial Monte Carlo.

A. TPU System Architecture

One TPU is comprised of four independent chips. Each chip consists of two compute cores called Tensor Cores. A Tensor Core, as shown in Fig. 1 consists of scalar, vector and matrix units (MXU). In addition, 16 GB of on-chip High Bandwidth Memory (HBM) is associated with each Tensor Core for Cloud TPU v3 — its latest generation. Communication between Tensor Cores occurs through high-bandwidth interconnects. All computing units in each Tensor Core are optimized to perform vectorized operations. In fact, the main horsepower of a TPU is provided by the MXU which is capable of performing 128×128 multiply-accumulate operations in each cycle [8]. While its inputs and outputs are 32-bit floating point values, the MXU typically performs multiplications at the reduced precision of bfloat16 — a 16-bit floating point representation that provides better training and model accuracy than the IEEE half-precision representation for deep learning as it allocates more bits to the exponent and less to the mantissa.

B. Programming Model

Programming for TPUs is generally done through high-level Tensorflow API. When the program is run, a TensorFlow computation graph is generated and sent to the Cloud TPU over gRPC [1]. The Cloud TPU server compiles the computation graph just in time, partitions the graph into portions that can run on a Cloud TPU and those that must run on a CPU and generates Accelerated Linear Algebra (XLA) operations corresponding to the sub-graph that is to run on Cloud TPU. Next, the XLA compilier takes over and converts High Level Optimizer (HLO) operations that are produced by the TensorFlow server to binary code that can be run on Cloud TPU, including orchestration of data from on-chip memory to hardware execution units and inter-chip communication. Finally, the program binary is sent to the Cloud TPU for execution.

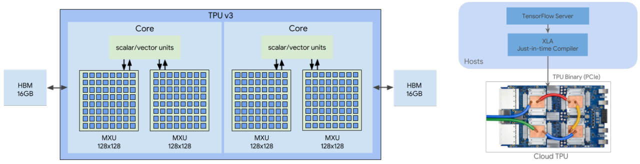


Fig. 1: Hardware architecture and programming model for Tensor Processing Units (TPUs). Detailed documentation is available at [1].

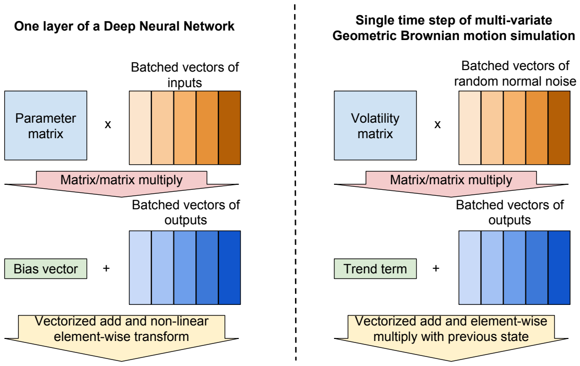

One layer of a Deep Neural Network
Batched vectors of inputs
Parameter matrix
x
Matrix/matrix multiply
Batched vectors of outputs
Bias vector
+
Vectorized add and non-linear element-wise transform
Single time step of multi-variate Geometric Brownian motion simulation
Batched vectors of random normal noise
Volatility matrix
x
Matrix/matrix multiply
Batched vectors of outputs
Trend term
+
Vectorized add and element-wise multiply with previous state

Fig. 2: Computational similarity between computing the output of a Deep Neural Network's layer and a single step of Monte Carlo simulation for correlated stochastic processes.

C. *Similarity between Deep Neural Network (DNN) inference and discretized correlated stochastic process simulation*

The main reason why we investigate the use of TPUs for risk assessment in quantitative finance is the computational similarity between DNN inference and simulating high dimensional financial portfolio. Tensorflow and TPUs have been designed to provide a high level interface for deep learning programming that is easy to use, is flexible and runs efficiently to train mission critical models rapidly. Our proposal is to leverage such flexibility and performance for a different purpose: running stochastic process simulation for financial applications.

From a computational standpoint, computing the output of DNN implies chaining matrix/matrix multiplies interleaved with element-wise vectorizable operations. Incidently, very similar computational patterns are involved in quantitative finance as instantiating Equation 4 requires the computation of matrix/matrix multiples (with batched PRNs) and element-wise vectorizable operations. Figure 2 illustrates this very close computational similarity. Furthermore, as training DNNs requires computing a gradient through automated differentiation between the outputs and all the parameters of the network, a DNN learning framework such as Tensorflow is an ideal candidate to enable sensitivity estimation through AAD in finance [31].

IV. Pseudo Random Number Generator (PRNG) on TPU

In the interest of brevity for this first approach to leverage TPUs for financial Monte-Carlo, we only consider PRNs and not Quasi-Random Numbers [16], [28] We describe how such numbers are generated on TPUs and the corresponding throughput.

A. *PRNGs: the old and the new*

In order to put the PRN generation procedure in perspective, we first recall principles employed in first generation PRN generators and introduce the key developments that enable simple and scalable implementations.

1) Classic PRNGs: For financial Monte Carlo, the Mersenne Twister 19337 (MT19337) [27] constitutes a very popular approach and yields a sequence of PRNs with period 2^{19337}. Such a PRN sequence is produced as a single stream and therefore parallelization of the sequence relies on sub-stream approaches. Sub-stream parallelization will either generate large non-overlapping sections of the sequence on each core or generate numbers with a skip-ahead to guarantee that distinct elements of the sequence are generated by each processor. While sub-stream parallelization produces correct PRN sequences, it has high computational costs, and it requires deciding ahead of time how many PRNs will be consumed by each core through the simulation which is not always possible.

2) PRNGs designed for parallelization: A more flexible approach for distributed computing is to use multi-stream random number generation as in [30]. Multi-stream algorithms, such as Threefry and Philox, use cryptographic methods to generate multiple sequences $(x_n^k)_{n=1 \dots N, k=1 \dots K}$ guaranteed to be seemingly statistically independent as long as each generating core employs a distinct key. The number of admissible keys is typically 2^{64} if 64 bit integers are used to encode keys and the period of each sequence is typically 2^{64} if 64 bits are used to represent the sequence iterator. [30] shows that these methods have higher throughput and better statistical properties when submitted to Big-Crunch [21] than standard generators. In particular, while Threefry and Philox pass all the tests in the battery of Big-Crunch, MT19937 fails most of them. We have confirmed that the generation of uniformly distributed 32 bit integers on TPU successfully passes Big-Crunch [21] (the double precision version of Big-Crunch is not directly relevant to TPUs which generate uniformly distributed floats in $[0, 1)$ in single precision only). For this reason and because of the ease of parallelization, Tensorflow relies on keyed multi-stream PRNGs. In particular, we employ Threefry in our TPU experiments and Philox on GPU.

V. Numerical Precision on TPU and Multi-Level Monte Carlo

Running dynamical system simulations, in particular in finance, often relies on high numerical precision to produce faithful results. We now delineate the native numerical precision of TPUs.

A. *Single and bfloat16 precision on TPU*

As opposed to today's CPUs or GPUs, TPUs do not offer double (64 bit) precision for floating point representations. The default representation precision is single (32 bit) precision and scalar multiplications in the MXU for matrix multiplies are computed in bfloat16 (16 bit) precision prior to being accumulated in single precision. As 16 bits are quite few to represent numbers for ML applications and chained multiplications in general, a non standard representation scheme has been employed to better capture high magnitude values and prevent overflow.

The single precision IEEE floating point standard allocates bits as follows: 1 sign bit, 8 exponent bits and 23 mantissa bits. The IEEE half precision standard uses 1 sign bit, 5 exponent bits and 10 mantissa bits. In contrast, the bfloat16 format employs 1 sign bit, 8 exponent bits and 7 mantissa bits.

We found in our numerical experiments that both the single precision — used in accumulators and vector units for element-wise operations — and the bfloat16 precision did not yield significantly different results as compared to the double precision in the context of financial Monte Carlo.

B. *Numerical precision, discretization bias and variance in risk metrics*

Financial Monte-Carlo methods are typically concerned with the simulation of a continuous time SDE such as Equation (1). Analyzing the convergence of the Monte-Carlo estimator $\widehat{I}_N$ in Equation (3) hinges upon the well known bias-variance decomposition of the L_2 error [28]:

$$||E\left[f(X_{0:T})\right]-\widehat{I}_N||_2^2 = \left(E\left[f(X_{0:T})\right] - E\left[\widehat{I}_N\right]\right)^2 + \frac{\text{Var}\left(\widehat{I}_N\right)}{N}.$$

The bias term typically conflates the bias induced by temporal discretization and floating point numerical representation.

When simulating a SDE, a temporal discretization occurs that induces most of the bias affecting the Monte-Carlo simulation. Indeed, as opposed to the actual process of interest $(X_t)_{t\in[0,T]}$, the Monte-Carlo simulation typically employs a piece-wise continuous approximation $(\widetilde{X}_t)_{t\in[0,T]}$ for which only H values are computed (if H is the number of temporal discretization steps) every Δ_t (where Δ_t is the temporal discretization step). Under several assumptions which are typically true for financial pricing, the Taley-Tubaro theorem [28] states that the discretization bias reduces as $O(\frac{1}{H})$, that is inversely to the number of steps employed when discretizing the process. Such a bias term dominates in practice as we will demonstrate in some of our numerical experiments. In comparison, the numerical precision induced bias is negligible. Furthermore, the impact of the estimator's variance on the L_2 error shows that adding more Monte Carlo paths is generally more beneficial in reducing it than increasing the numerical precision. Finally, the MLMC method can efficiently estimate the numerical bias term if it is significant and compensate for it.

VI. NUMERICAL EXPERIMENTS

We now demonstrate that TPUs are fast instruments whose numerical precision is commensurate with the needs of financial Monte-Carlo based risk metrics. In the following, we use the Philox on GPU and Threefry on TPU (where it is more challenging to support certain low-level operations leveraged by Philox). Also, we use the same python Tensorflow code on GPU and TPU for two reasons: we want to make our comparison with a library that has been highly optimized for speed and we want to make comparisons of speed at equal level of software engineering effort. Here NVidia v100 GPUs are used to anchor our results, but no architecture specific optimizations have been done, which could considerably improve their performance. Therefore, the GPU wall times we give can only be considered a solid reference but not an indicator of the peak performance one can obtain on the NVidia V100 GPU. In the following, we do not take into account compilation times when measuring wall time on TPU as the corresponding overhead is obviously amortized in most applications that are latency sensitive, and is not a hindrance for an interactive experience in a Colaboratory notebook (interactive notebooks comparable to ipython Jupyter notebooks).

A. *European option pricing and hedging*

Our first experiments are concerned with European option pricing, i.e. non-callable derivatives with a single terminal payoff.

1) Uni-variate process simulation to benchmark TPUs' VPU: Uni-variate stochastic process simulations do not need matrix multiplies and therefore constitute helpful benchmarks to specifically assess the performance of the Vector Processing Unit (VPU) on TPUs, which performs single (32-bit) floating point arithmetic.

Vanilla Call: First we start with the extremely simple example of pricing a 1 year maturity European Call option (strike at 120) under the standard Black-Scholes model with constant drift (0.05) and volatility (0.2) with an initial underlying price of 100. The analytic price of the option in double precision is 3.24747741656. What we intend to show here — as we know the actual option price analytically — is that temporal discretization bias largely dominates over numerical precision bias. Each of the 100 simulation runs has 25 to 100 discretization steps and 10M samples. One can verify in Figure 3 that TPUs provide a comfortable speed-up compared to GPUs running the same Tensorflow graph, and that the bias induced by the use of lower precision is negligible compared to that induced by the temporal discretization.

Path dependent exotic Put: We make things slightly more complex and consider pricing an In and Out Put (activating Barrier). The initial price of the underlying is 100, the strike 120, the barrier 140. The drift is still constant (0.03) as well as the volatility (0.8). The analytic price of the option (given by the symmetry principle under the Black-Scholes model) in double precision is 23.1371783926. Between each two discretization steps with values x and y we simulate the maximum of the Brownian bridge — classically [16], [28] with

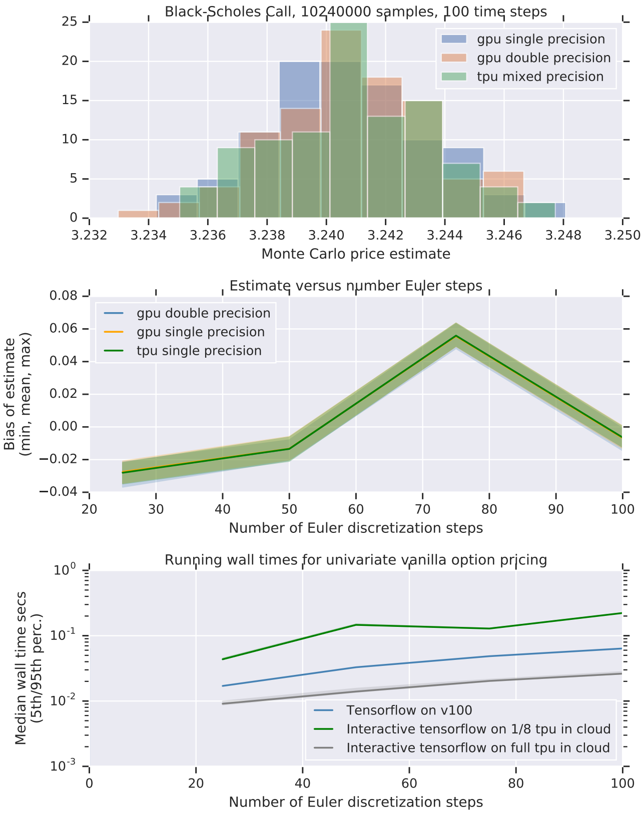


Fig. 3: **Vanilla call:** Estimate distributions in single and double precision. Bias terms with various numerical precision and discretization steps for the estimation of a European Call price under a Black-Scholes model (10^8 samples) and their sensitivity with respect to the temporal discretization step size. The temporal discretization bias clearly dominates the numerical precision bias.

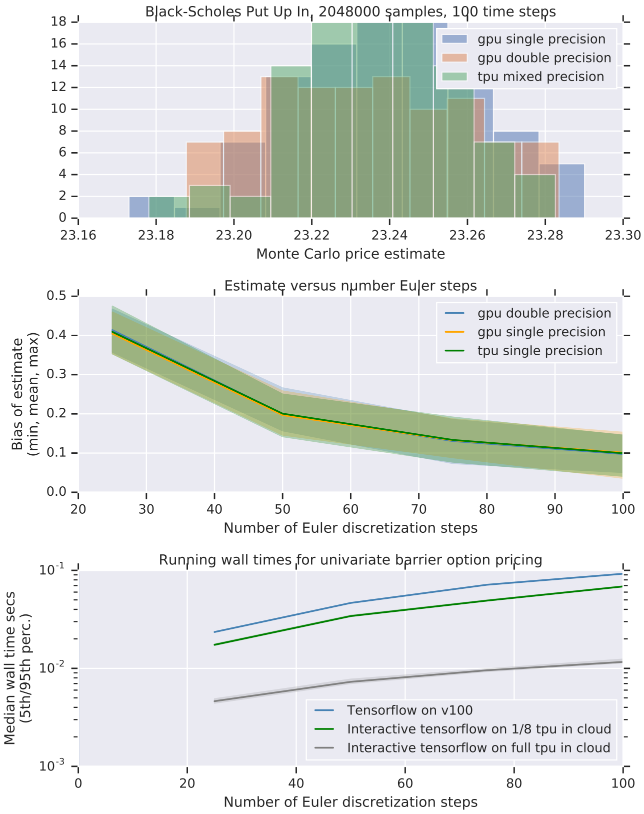


Fig. 4: **Path dependent exotic Put:** Estimate distributions in single and double precision. Bias terms with various numerical precision and discretization steps for the estimation of a European Put Up In price under a Black-Scholes model and their sensitivity with respect to the temporal discretization step size. Again, the temporal discretization bias clearly dominates the numerical precision bias.

$m \sim 0.5\left(x+y+\sqrt{\left((x-y)^2\right)-2\Delta t\sigma^2\log(U)}\right)$ where U is distributed uniformly in $(0,1)$ — to reduce the discretization bias of the method. Such an experiment also demonstrates that programming in Tensorflow is flexible enough to allow for varied simulation schemes. Each of the 100 simulation runs has 25 to 100 discretization steps and 2M samples. Again, in Figure 4, one can appreciate the speed up for a Tensorflow graph running on TPU with respect to an identical graph running on GPU while the impact of the use of single precision as opposed to double precision is hardly noticeable.

2) Multi-variate process simulation to benchmark TPUs' MXU: Multi-variate stochastic simulations represent heavier computational workloads than their uni-variate counterparts whenever they involved correlated dynamics. A matrix/matrix multiply is then involve at each step of the simulation when computing the product of the volatility matrix with the multidimensional normally distributed stacked PRNs. In such a setting, the speed of the MXU on TPUs may be beneficial, but, as it uses a custom floating point representation, one needs to assess that no substantial numerical precision bias appears.

Basket European option: We price an at-the-money Basket European call with 2048 underlyings whose price is initially 100. The interest rate is 0.05 and the volatility matrix we use is a historical estimate based on market data collected on daily variations of randomly selected stocks from the Russell 3000 through 2018. 2K samples are used for each of the 100 simulations. Simulations now involve matrix multiplications corresponding to Equation (1) and therefore the MXU of the TPU is used with a reduced bfloat16 precision. All other computations on TPU run in single precision. In Figure 5 we present the estimates provided in mixed precision on TPU and compare them with single and double precision estimates. We find that running simulations on TPU does not introduce any significant bias while offering substantial speed ups compared to GPUs.

Basket European option Delta: As automated differentiation is integrated into the Tensorflow framework, almost no additional engineering effort is required to compute pathwise sensitivities of a MC simulation. Considering the same

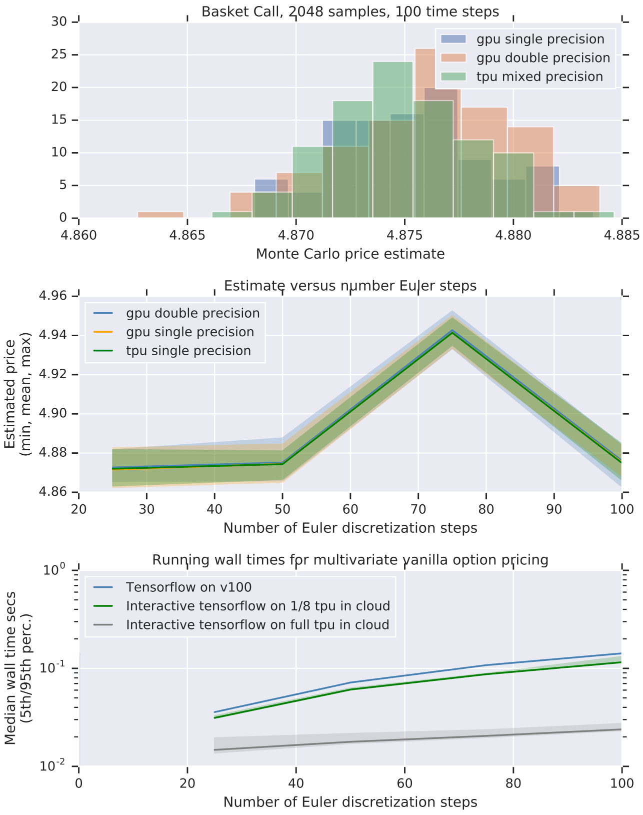


Fig. 5: **Basket European Option:** Estimate distributions and wall times when the MXU is employed on TPU with mixed precision. No significant bias is introduced by running computations on a TPU. The measured wall times, which include the network round trip between front end host and TPU, are very competitive.

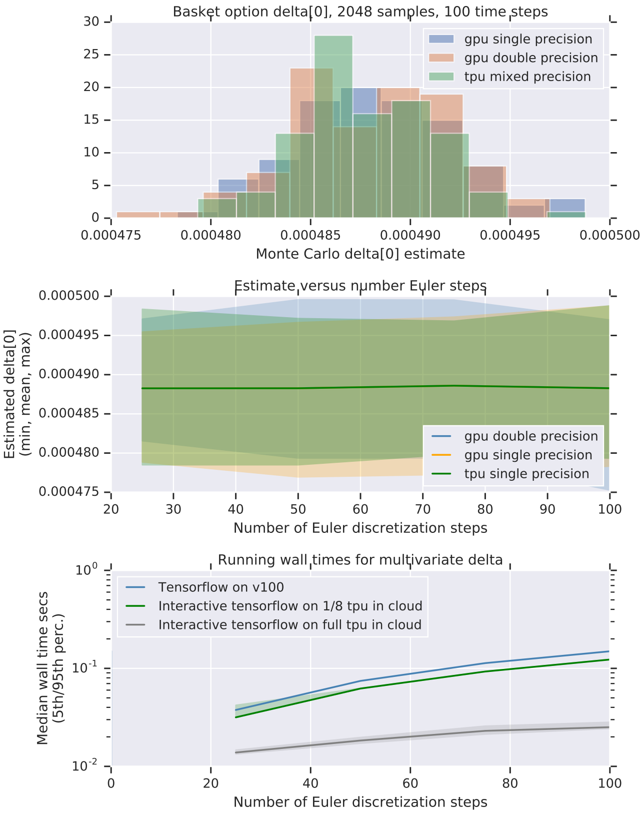


Fig. 6: **Basket European Option Delta:** Estimate distributions and wall times for the first component of the option's delta when the MXU is employed on TPU with mixed precision. No significant bias is introduced by running computations on a TPU. The measured wall times, which include the network round trip between front end host and TPU, are very competitive.

European option, we now compute its "delta", that is to say the first order derivative of the option's price estimate with respect to the initial price vector (with 2048 components). As demonstrated in the code snippet of Figure 10, presented in section VII, computing such a sensitivity estimate, which can represent a substantial software engineering effort for libraries not designed with AAD in mind [31], only requires a single line of code in Tensorflow. In Figure 6, we can appreciate that although back-propagation introduces an additional chain of multiplications, no significant bias is added (we present results here only for the first component of the "delta" for ease of reading).

B. Risk metrics

The impact of mixed precision on variance may become a concern for risk metrics such as VaR and CVaR whose purpose is to estimate percentiles and losses for a given portfolio that occur in rare adverse scenarios.

1) Estimating Value-at-Risk with many underlying factors: In this simulation, we consider the simulation of the same 2048 underlying assets as in the Basket option experiment (all from the Russel 3000) with a trend and correlation structure estimated based on historical data on daily variations. The portfolio whose loss distribution we sample from consists of 5 call options with different strikes on each of the underlying assets. As a result the portfolio of interest has 10240 instruments.

Value-at-Risk: We simulate the distribution of profit and losses (PnL) for the portfolio of interest over the course of a year with different scales of temporal discretization. The first risk metric of interest is the standard Value-at-Risk (VaR) at level α. By definition, VaR is a quantile of the distribution of losses on an investment in the presence of contingencies. Given a random variable $\text{PnL}(\omega)$ representing the PnL of the overall portfolio subjected to random perturbations ω, we have to estimate the quantile of level α of the PnL distribution:

$$\text{VaR}_\alpha\left(\text{PnL}(\omega)\right) = -\inf\left\{x \in \mathbb{R} : \mathbb{P}\left(\text{PnL}(\omega) \geq x\right) > \alpha\right\}.$$

The results presented in Figure 7 show that limited precision in the MXU has some impact in the estimated VaR terms as some bias is present (less than 1% in relative magnitude

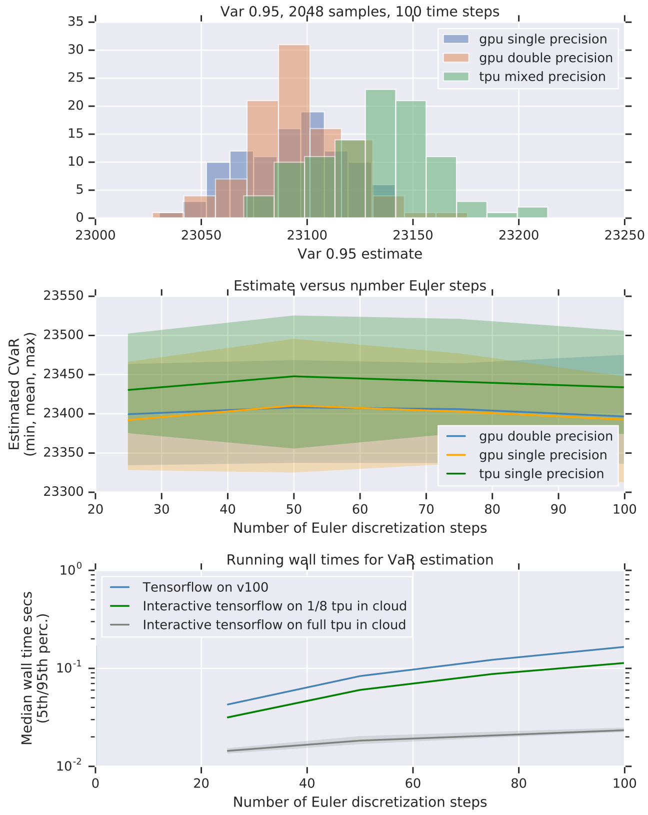


Fig. 7: **Value-at-Risk:** Estimates of $\text{VaR}_{0.95}$ and corresponding wall times (the MXU is used with bfloat16 precision on TPU for matrix multiplies). Some bias is introduced by running computation in mixed precision on TPU. The measured wall times (including the network round trip for TPUs) are very competitive for TPU.

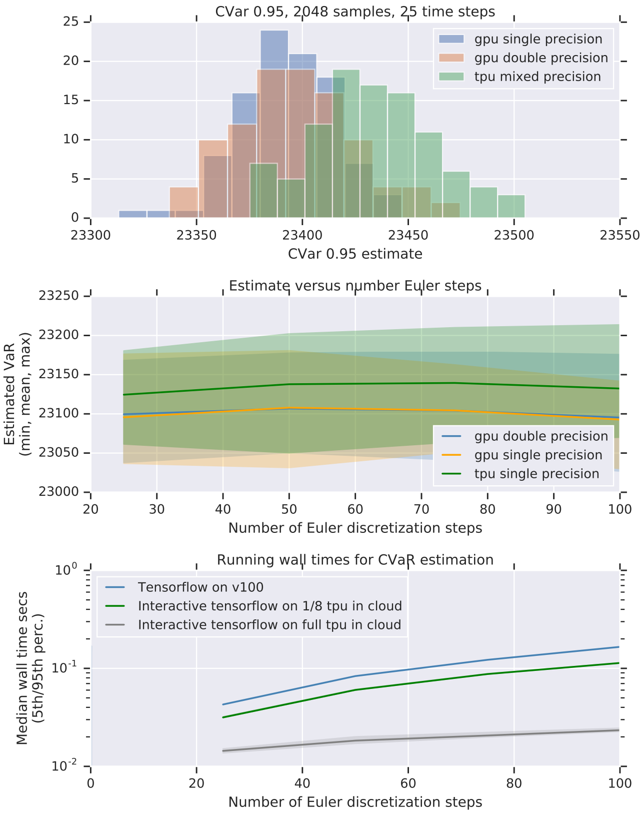


Fig. 8: **Conditional Value-at-Risk:** Estimates of $\text{CVaR}_{0.95}$ and corresponding wall times (the MXU is used with bfloat16 precision on TPU for matrix multiplies). Some bias is introduced by running computation in mixed precision on TPU. The measured wall times (including the network round trip for TPUs) are very competitive for TPU.

compared to a double precision simulation). However, the speed-ups reported are substantial, so that a Multi-Level-Monte-Carlo approach could be employed to preserve most of the computational gains while identifying the TPU-induced bias to later correct it.

Conditional Value-at-Risk: The Conditional Value-at-Risk (CVaR) (otherwise known as expected shortfall) is another risk metric used jointly with VaR. The great advantage of CVaR is that it is a coherent risk measure and therefore provides a more principled view on risks associated with a given portfolio [16]. While VaR is defined as the quantile of a loss distribution, CVaR corresponds to a conditional expectation on losses. More precisely, CVaR estimates the expected loss conditioned to the fact that the loss is already above the VaR. For a level of tolerance α, CVaR_α is defined as follows:

$$\text{CVaR}_\alpha\left(\text{PnL}(\omega)\right) = -E_{\mathbb{P}}\left[PnL(\omega)\right| - PnL(\omega) > \text{VaR}_\alpha\right].$$

The results reported in Figure 8 demonstrate that while the use of mixed precision on TPU introduces some bias (less than 1% in relative magnitude compared to a double precision simulation) in the estimation of CVaR it also comes with substantial speed ups. As for the computation of VaR, the results indicate that TPUs are therefore good candidates for the use of Multi-Level-Monte-Carlo to produce unbiased estimates rapidly.

C. Monte Carlo American option pricing

Monte Carlo option pricing of an American option with multiple underlyings presents the computational difficulties encountered in European Basket option pricing because of the need for the simulation of multi-variate diffusion processes while adding the additional complexity of having to proceed with dynamic programming.

1) Longstaff-Schwartz pricing of an American Maximum Option: The Longstaff-Schwartz (LSM) method for multi-variate American option pricing relies on a set of samples of the paths of the underlyings to compute the terminal payoff at the expiration of the option assuming there was no early exercise. As explained earlier in sub-section II-A3, LSM then proceeds with dynamic programming taking the form of chained linear regressions by Ordinary Least Squares. In pratice, we use the Cholesky based solver for such systems

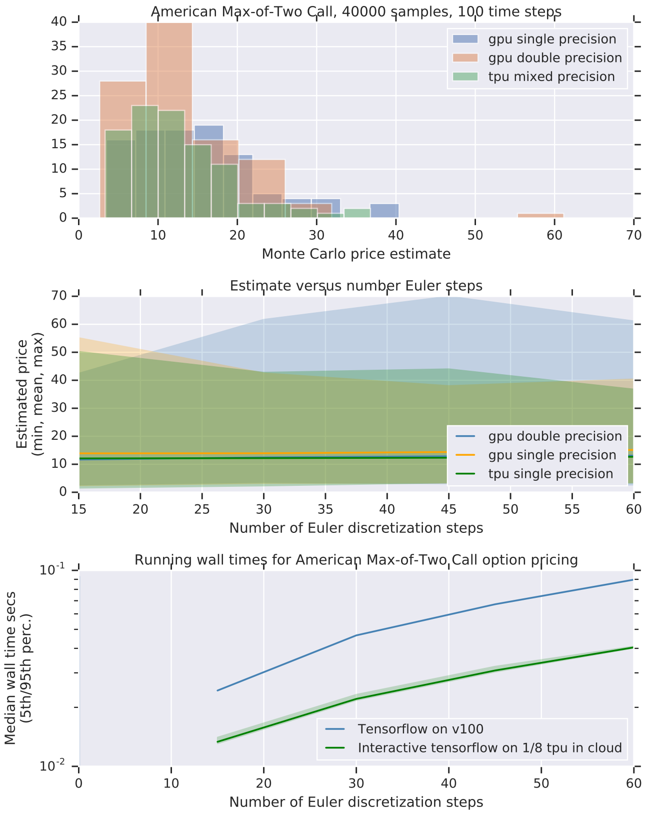
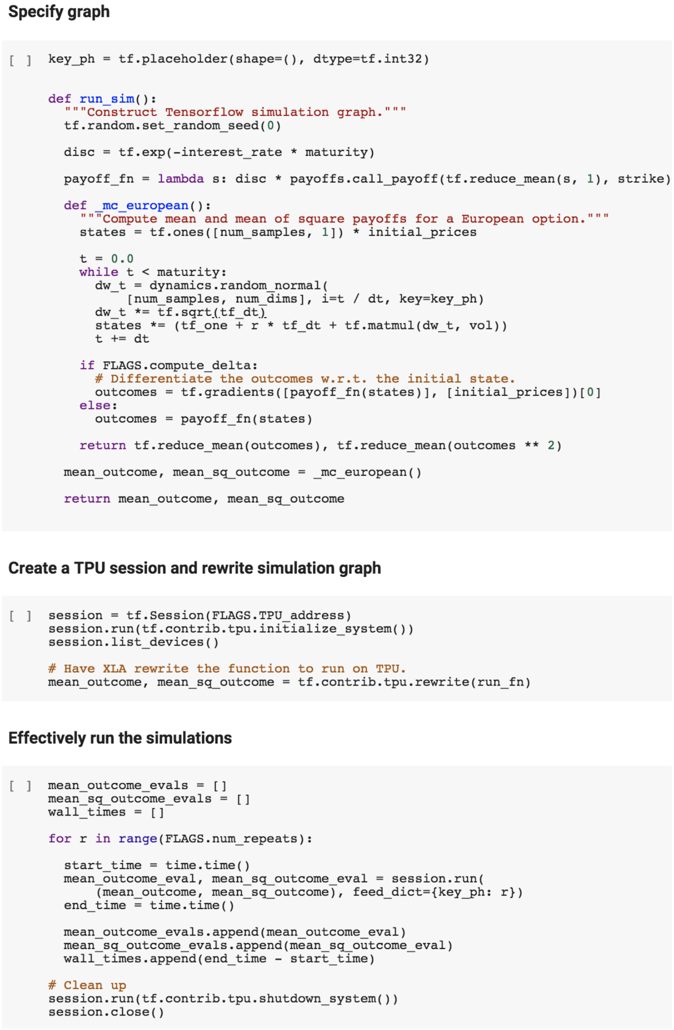


Fig. 9: **Longstaff-Schwartz pricing of an American Maximum Option:** LSM is employed on both TPU and GPU to price an American Max-of-Two Call. The very setting of Example 8.6.1 from [16] is reproduced and the estimates on all hardware are close to the stated true price of 13.90.

integrated in Tensorflow after a Cholesky decomposition of the Grammian. From a numerical standpoint, as the MXU is now used for both the forward simulations and the linear inversions in dynamic programming there is an added level of uncertainty related to the impact of the use of bfloat16 in the MXU. The mixed precision could be causing numerical instability problems. In practice, this has not been the case on TPU and we have had to add a regularization term to the regression only to help the GPU implementation which was often failing to do the Cholesky decomposition. We reproduced the experiment reported in [16] in Example 8.6.1 which assesses the true price as 13.90. The results presented in Figure 9 show that no significant bias was introduced by the use of TPU while substantial speedups were gained. It is noteworthy that here we only use one TPU core because there are multiple ways of parallelizing LSM and discussing their different properties is beyond the scope of the present paper.


Specify graph

```
[ ]  key_ph = tf.placeholder(shape=(), dtype=tf.int32)


     def run_sim():
       """Construct Tensorflow simulation graph."""
       tf.random.set_random_seed(0)

       disc = tf.exp(-interest_rate * maturity)

       payoff_fn = lambda s: disc * payoffs.call_payoff(tf.reduce_mean(s, 1), strike)

       def _mc_european():
         """Compute mean and mean of square payoffs for a European option."""
         states = tf.ones([num_samples, 1]) * initial_prices

         t = 0.0
         while t < maturity:
           dw_t = dynamics.random_normal(
               [num_samples, num_dims], i=t / dt, key=key_ph)
           dw_t *= tf.sqrt(tf_dt)
           states *= (tf_one + r * tf_dt + tf.matmul(dw_t, vol))
           t += dt

         if FLAGS.compute_delta:
           # Differentiate the outcomes w.r.t. the initial state.
           outcomes = tf.gradients([payoff_fn(states)], [initial_prices])[0]
         else:
           outcomes = payoff_fn(states)

         return tf.reduce_mean(outcomes), tf.reduce_mean(outcomes ** 2)

       mean_outcome, mean_sq_outcome = _mc_european()

       return mean_outcome, mean_sq_outcome
```

Create a TPU session and rewrite simulation graph

```
[ ]  session = tf.Session(FLAGS.TPU_address)
     session.run(tf.contrib.tpu.initialize_system())
     session.list_devices()

     # Have XLA rewrite the function to run on TPU.
     mean_outcome, mean_sq_outcome = tf.contrib.tpu.rewrite(run_fn)
```

Effectively run the simulations

```
[ ]  mean_outcome_evals = []
     mean_sq_outcome_evals = []
     wall_times = []

     for r in range(FLAGS.num_repeats):

       start_time = time.time()
       mean_outcome_eval, mean_sq_outcome_eval = session.run(
           (mean_outcome, mean_sq_outcome), feed_dict={key_ph: r})
       end_time = time.time()

       mean_outcome_evals.append(mean_outcome_eval)
       mean_sq_outcome_evals.append(mean_sq_outcome_eval)
       wall_times.append(end_time - start_time)

     # Clean up
     session.run(tf.contrib.tpu.shutdown_system())
     session.close()
```

Fig. 10: Code snippet to set up a simulation in an interactive notebook.

VII. PROGRAMMING SIMULATIONS WITH TENSORFLOW

A. Minimal code to run a Monte Carlo simulation

In Figure 10, we show the sufficient code to set up a Monte Carlo simulation in Tensorflow. We demonstrate effectively that one can devise a simulation for a European Basket Option price estimator with a few lines of code, all while working in an interactive Colaboraty (or Jupyter) notebook.

B. Automated Differentiation with one line of code

In Figure 10, we also show that a single line of code suffices to turn payoff estimates into sensitivies (first order derivatives with respect to the initial price in this case computed by AAD, i.e. back-propagation). This is a remarkable consequence of employing the Tensorflow framework which is not only optimized for linear algebra acceleration as well as fast random number generation but also integrates automated differentiation.

VIII. Conclusion

In conclusion we argue that TPUs are indeed accurate, fast and easy to use for financial simulation. Our experiments on multiple diverse workloads demonstrate that even for large simulations written in Tensorflow, TPUs enable a responsive interactive experience with a higher speed than GPUs running Tensorflow. Results also indicated that if cases arise in which the mixed precision calculations running on TPUs create too significant a bias, Multi-Level-Monte-Carlo offer an efficient way of correcting this bias. As next steps, we want to confirm Multi-Level-Monte-Carlo methods can work successfully on TPUs. We also aim to enable faster model calibration and leverage more advanced models for better derivatives pricing and improved risk assessment.

References

[1] Cloud tpu documentation. https://cloud.google.com/tpu/docs/. Accessed: 2019-04-29.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[3] Abbas-Turki, L. A., Vialle, S., Lapeyre, B., and Mercier, P. Pricing derivatives on graphics processing units using monte carlo simulation. *Concurrency and Computation: Practice and Experience 26*, 9 (2014), 1679–1697.

[4] Bardou, O., Frikha, N., and Pages, G. Computing var and cvar using stochastic approximation and adaptive unconstrained importance sampling. *Monte Carlo Methods and Applications 15*, 3 (2009), 173–210.

[5] Bergomi, L. *Stochastic volatility modeling*. CRC Press, 2015.

[6] Brugger, C., de Schryver, C., and Wehn, N. Hyper: a runtime reconfigurable architecture for monte carlo option pricing in the heston model. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), IEEE, pp. 1–8.

[7] Chow, G. C. T., Tse, A. H. T., Jin, Q., Luk, W., Leong, P. H., and Thomas, D. B. A mixed precision monte carlo methodology for reconfigurable accelerator systems. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (2012), ACM, pp. 57–66.

[8] Cloud, G. *Performance Guide*. https://cloud.google.com/tpu/docs/performance-guide, 2019.

[9] Coddington, P. D. Random number generators for parallel computers.

[10] Dixon, M. F., Bradley, T., Chong, J., and Keutzer, K. Monte carlo–based financial market value-at-risk estimation on gpus. In *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 337–353.

[11] Du Toit, J., Lotz, J., and Naumann, U. Adjoint algorithmic differentiation of a gpu accelerated application, 2013.

[12] Dupire, B., et al. Pricing with a smile. *Risk 7*, 1 (1994), 18–20.

[13] Gatheral, J., Jaisson, T., and Rosenbaum, M. Volatility is rough. *arXiv preprint arXiv:1410.3394* (2014).

[14] Giles, M. B. Multilevel monte carlo path simulation. *Operations Research 56*, 3 (2008), 607–617.

[15] Giles, M. B. Multilevel monte carlo methods. *Acta Numerica 24* (2015), 259–328.

[16] Glasserman, P. *Monte Carlo methods in financial engineering*, vol. 53. Springer Science & Business Media, 2013.

[17] Glasserman, P., and Li, J. Importance sampling for portfolio credit risk. *Management science 51*, 11 (2005), 1643–1656.

[18] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

[19] Guyon, J., and Henry-Labordere, P. The smile calibration problem solved.

[20] Jouppi, N. P. *Quantifying the performance of the tpu, our first machine learning chip.* https://cloud.google.com/blog/products/gcp/quantifying-the-performance-of-the-tpu-our-first-machine-learning-chip, 2017.

[21] L'Ecuyer, P., and Simard, R. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS) 33*, 4 (2007), 22.

[22] Lee, A., Yau, C., Giles, M. B., Doucet, A., and Holmes, C. C. On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of computational and graphical statistics 19*, 4 (2010), 769–789.

[23] Lemaire, V., Pagès, G., et al. Unconstrained recursive importance sampling. *The Annals of Applied Probability 20*, 3 (2010), 1029–1067.

[24] Longstaff, F. A., and Schwartz, E. S. Valuing american options by simulation: a simple least-squares approach. *The review of financial studies 14*, 1 (2001), 113–147.

[25] LEcuyer, P., Munger, D., Oreshkin, B., and Simard, R. Random numbers for parallel computers: Requirements and methods, with emphasis on gpus. *Mathematics and Computers in Simulation 135* (2017), 3–17.

[26] Marshall, T. J., Reesor, R. M., and Cox, M. Simulation valuation of multiple exercise options. In *Proceedings of the Winter Simulation Conference* (2011), Winter Simulation Conference, pp. 3772–3783.

[27] Matsumoto, M., and Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[28] Pagès, G. *Numerical Probability: An Introduction with Applications to Finance*. Springer, 2018.

[29] Podlozhnyuk, V., and Harris, M. Monte carlo option pricing. *CUDA SDK* (2008).

[30] Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 16.

[31] Savine, A. *Modern Computational Finance: AAD and Parallel Simulations*. Wiley, 2018.

[32] Thomas, D. B., Howes, L., and Luk, W. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (2009), ACM, pp. 63–72.

[33] Weston, S., Marin, J.-T., Spooner, J., Pell, O., and Mencer, O. Accelerating the computation of portfolios of tranched credit derivatives. In *2010 IEEE Workshop on High Performance Computational Finance* (2010), IEEE, pp. 1–8.

[34] Weston, S., Spooner, J., Racanière, S., and Mencer, O. Rapid computation of value and risk for derivatives portfolios. *Concurrency and Computation: Practice and Experience 24*, 8 (2012), 880–894.