Monte Carlo option pricing in tensorflow

Kai Detlefsen

June 11, 2018

Abstract

We look at pricing financial derivatives on gpus. After some general comparison of frameworks, we focus on pricing via Monte Carlo simulations and compare numpy, tensorflow cpu and tensorflow gpu using the Black Scholes model, the Heston model and the Heston model with local volatility.

1 Introduction

Financial derivatives are products whose values depend on some underlying prices. These products can be priced analytically, by pde or by Monte Carlo simulation. We consider the pricing by Monte Carlo which is often used when the models are advanced and there are many underlying price processes. This applies eg to exotic derivatives in equity.

Around 2008, several banks implemented Monte Carlo pricing on gpus by using cuda. Then the focus shifted to other topics. Now there is a general interest in using gpus for neural networks eg in natural language processing or computer vision. There is also some interest in pricing derivatives on gpus because simpler and more powerful frameworks have become available. We look here at the tensorflow framework and make some tests to get a feeling for the speedup and for implementation complexity. For comparison, we use numpy which is based - like tensorflow - on an implementation in C++.

2 Basic operations

Let us look first how gpus handle basic operations focusing on matrix multiplication. We use a laptop with an i5-7200 processor 8GB and an nvidia gtx950m gpu 4GB. This gpu is on the lower end of the nvidia gpus. Moreover, we use numpy 1.14 and tensorflow 1.7 with cuda 8. The numpy version uses the Intel math kernel library while we didnt make a tensorflow build particularly for the processor. The code of the examples can be found here. They use float32 numbers in numpy and tensorflow in order to make times more comparable.

The timings in numpy are computed with the following code:

```
A = np.random.randn(n, n).astype(np.float32)
B = np.random.randn(n, n).astype(np.float32)
t = timeit.repeat("np.matmul(A, B)", "import numpy as np;
      from __main__ import A, B", repeat=num_repeats, number=1)
```

We use matrices A and B of size n filled with standard Gaussians. The *timeit* module is used to time repeated iterations of $np.matmul(A, B)$.

In tensorflow, you first build the computation graph and then evaluate it. Here is the code that multiplies in tensorflow the matrices that were created in numpy:

```
with tf.device("/cpu:0"):
    C = tf.constant(A)
    D = tf.constant(B)
    CD = tf.matmul(C, D)
```

The graph is constructed by defining the constants C and D and their multiplication $tf.matmul(C, D)$. Tensorflow has some general rules for placing calculations on devices, eg cpu or gpu. Here, we pinned the graph to the cpu. It can be evaluated by

```
with tf.Session() as sess:
    t = timeit.repeat("sess.run(CD)", setup="import tensorflow as tf;
          from __main__ import sess, A, B, CD", repeat=num_repeats,
          number=1)
```

The graph can be pinned on a gpu by specifying "/gpu:0" as device.

When we take the times for one repetition, then we get the numbers in table 1. The multiplication on the gpu appears to be slow. One reason for this is that we pass the matrices from numpy to tensorflow which copies the data from the cpu memory to the gpu memory. Lets check the times when we create the random numbers directly on the gpu with the following code

n	numpy	tf cpu	tf gpu
625	0.026	0.037	0.044
1250	0.028	0.152	0.169
2500	0.267	1.411	1.237
5000	1.71	10.09	10.33

Table 1: run time (in s) of matmul for one repetition

```
G = tf.random_normal([n, n])
H = tf.random_normal([n, n])
GH = tf.matmul(G, H)
```

In this case, we get the run times of table 2 which shows that the gpu time indeed goes down but still is not fast compared to numpy.

n	numpy	tf cpu	tf gpu	tf gpu2
625	0.026	0.037	0.044	0.345
1250	0.028	0.152	0.169	0.683
2500	0.267	1.411	1.237	0.260
5000	1.71	10.09	10.33	1.877

Table 2: run time (in s) of matmul for one repetition where gpu2 creates the Gaussians on the gpu

In order to distinguish between random number generation and matrix multiplication on gpu, let us look at the timeline in detail with the following code

```
from tensorflow.python.client import timeline
with tf.Session() as sess:
    options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
    run_metadata = tf.RunMetadata()
    sess.run(GH, options=options, run_metadata=run_metadata)
    fetched_timeline = timeline.Timeline(run_metadata.step_stats)
    chrome_trace = fetched_timeline.generate_chrome_trace_format()
    with open('timeline_Matmul.json', 'w') as f:
        f.write(chrome_trace)
```

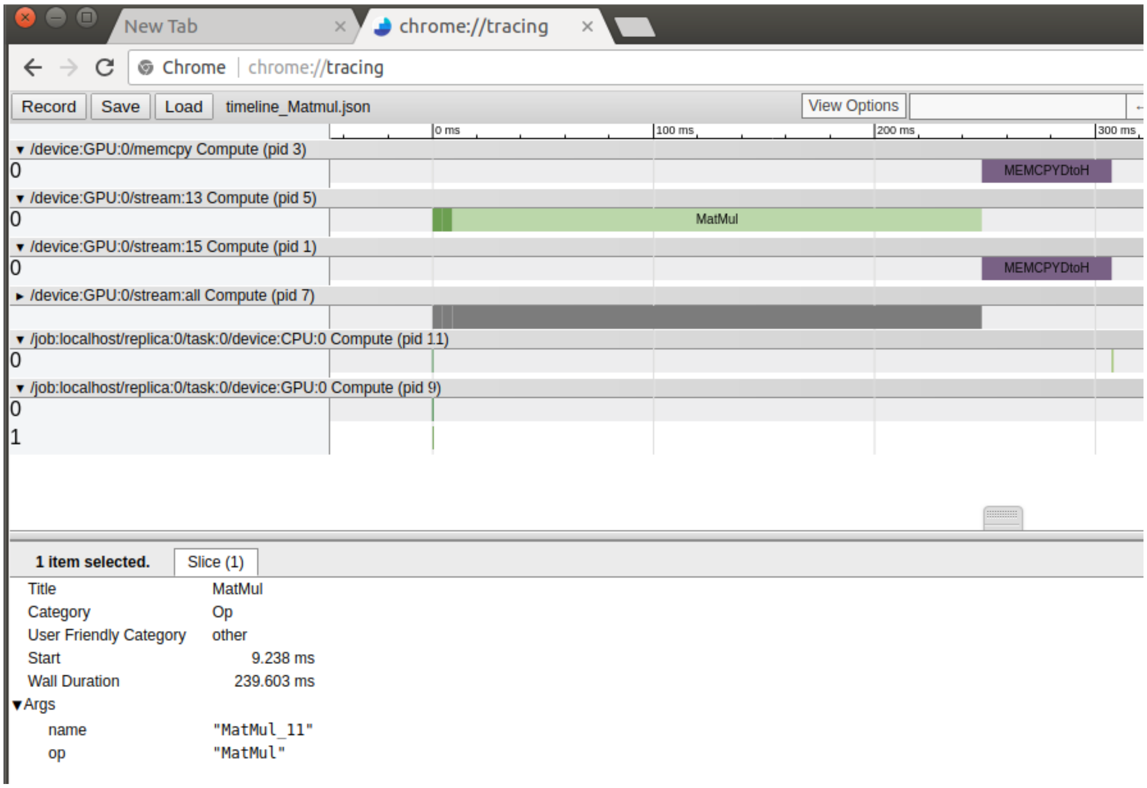
This code creates a log with details which can be inspected in chrome. For $n = 5000$ repetitions, we see the timeline in figure 1. This shows that the creation of the Gaussian takes less than 10ms on the gpu and the matrix multiplication takes 0.24s.

The timeline shows a fast matrix multiplcation while our timing shows a slow multiplication. Why is this? If we increase the number of repetition, then we see the fast multiplication in table 3 which is also illustrated in figure 2. Hence tensorflow spends some time for initialization in the first run. Moreover, figure 2 shows that 0.03s are used for memory copying. This is relatively small compared to the time for the matrix multiplication.

n	numpy2	tf cpu2	tf gpu3	tf gpu4
625	0.004	0.000	0.001	0.002
1250	0.028	0.001	0.004	0.008
2500	0.208	0.300	0.045	0.047
5000	1.595	1.587	0.298	0.304

Table 3: run times (in s) of matmul show the minimal time over two repetitions (gpu4 creates the Gaussians on the gpu)

For Monte Carlo simulations, elementwise operations are also important. As an example, we look at the exponential function which is on the gpu faster than the copying of the memory. Hence we consider in the comparison also the pure


New Tab
chrome://tracing
Chrome
chrome://tracing
Record
Save
Load
timeline_Matmul.json
View Options
0 ms
100 ms
200 ms
300 ms
/device:GPU:0/memcpy Compute (pid 3)
0
MEMCPYDtoH
/device:GPU:0/stream:13 Compute (pid 5)
0
MatMul
/device:GPU:0/stream:15 Compute (pid 1)
0
MEMCPYDtoH
/device:GPU:0/stream:all Compute (pid 7)
/job:localhost/replica:0/task:0/device:CPU:0 Compute (pid 11)
0
/job:localhost/replica:0/task:0/device:GPU:0 Compute (pid 9)
0
1
Slice (1)

1 item selected.

Title	MatMul
Category	Op
User Friendly Category	other
Start	9.238 ms
Wall Duration	239.603 ms
▼Args	
name	"MatMul_11"
op	"MatMul"

Figure 1: timeline of tf operations for creation and multiplication of Gaussians

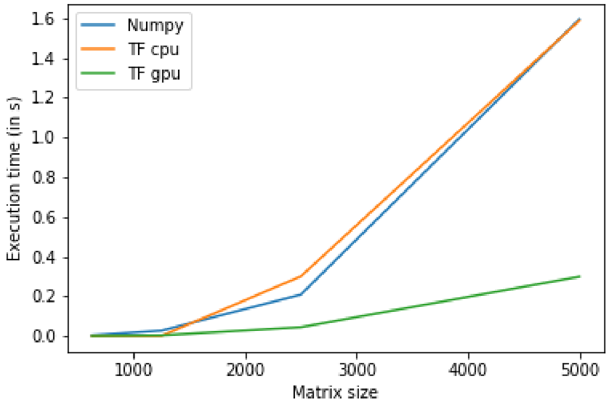


Figure 2: run times (in s) of matmul

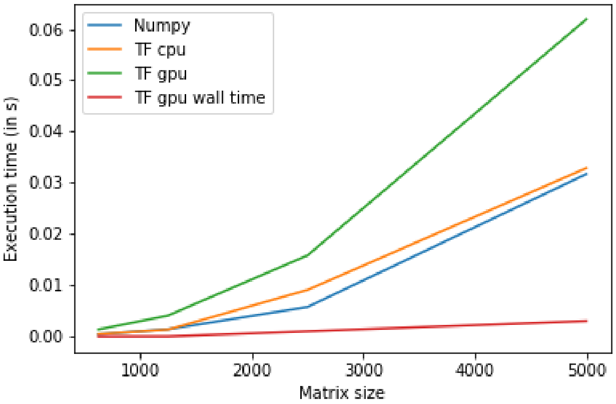


Figure 3: run times (in s) of exponential

wall time which can be seen in the timeline. The results are presented in figure 3. This shows that long calculations performed exclusively on the gpu without copying memory use the benefits of the gpu very well.

3 Models

We implement three models for the stock price (S_t): Black Scholes, Heston and Heston with local volatility. The Black Scholes is a one factor model, the Heston model has two factors and the extension to include local volatility adds the complexity to compute conditional expected values. We test the implementations of these models and look the run times for pricing an Asian option with payoff

$$\max(\frac{1}{n}\sum_{i=1}^{n} S_{t_i} - 1, 0)$$

where we consider daily averaging over a year, ie $n = 365$ and $t_i = i/365$.

The Black Scholes model is given by

$$\frac{dS_t}{S_t} = \sigma dW_t$$

where (W_t) is a Wiener process and $\sigma > 0$ is the volatility.

We give an implementation of a simple Monte Carlo scheme in numpy where we denote the number of simulations by $nSim$. We first draw standard Gaussians to simulate the returns R which give the stock values S. From this we compute the Asianing A and estimate the price of the option P.

```
R = sigma*sqrtDt*np.random.randn(nT, n) - 0.5*sigma*sigma*dt
C = np.cumsum(R, axis=0)
S = np.exp(C)
A = np.mean(S, axis=0)
P = np.mean(np.maximum(A - K, 0))
```

The numpy implementation can be converted directly to tensorflow for constructing the graph.

```
with tf.device(tf_device_name):
    n = tf.placeholder(tf.int32)
    Rt = sigma*sqrtDt*tf.random_normal((nT, n)) - 0.5*sigma*sigma*dt
    Ct = tf.cumsum(Rt, axis=0)
    St = tf.exp(Ct)
    At = tf.reduce_mean(St, axis=0)
    Pt = tf.reduce_mean(tf.maximum(At - K, 0))
```

We use a placeholder for the number of simulations such that we can change this parameter without having to rebuild the graph. The graph can be evaluated in a session as described before. In order to switch between cpu and gpu, we only have to specify a different device name, tensorflow takes care of the rest.

We compare the run time for pricing the Asian option via the above implementations in numpy, tensorflow cpu and tensorflow gpu. Tensorflow calculations have some fixed cost like constructing the graph and initializing the session. We ignore these steps for the comparison of run times. Such fixed costs play no role when a whole portfolio of trades is considered.

Table 4 presents the results. Tensorflow cpu is faster than numpy although numpy uses MKL. This might due to optimizations by tensorflow for the whole graph. Tensorflow gpu is faster than numpy and the speed advantage grows with the number of simulations reaching a speedup of factor 50 for 250000 simulations.

num sims	numpy	cpu	gpu
10000	0.24	0.14	0.022
25000	0.53	0.32	0.029
50000	1.07	0.61	0.038
100000	2.14	1.18	0.055
250000	5.37	2.97	0.114

Table 4: run times (in s) for pricing an Asian option in the Black Scholes model via numpy, tensorflow cpu and gpu

Next, we consider the Heston model. This model has more parameters and hence can fit better market data. The model is given by

$$\frac{dS_t}{S_t} = \sqrt{V_t} dW_t^1$$
$$dV_t = \kappa(\theta - V_t)dt + \alpha\sqrt{V_t} dW_t^2$$

with Wiener processes W^1 and W^2 correlated by ρ.

The numpy implementation of the Black Scholes model can be extended to the Heston model

```
W1 = np.random.randn(nT, n)
W2 = rhobar*np.random.randn(nT, n) + rho*W1

V = np.full((nT, n), V0)
for i in range(1, nT):
    V[i,:] = np.fabs(V[i-1,:] + kappa * (theta - V[i-1,:])*dt +
        alpha * np.sqrt(V[i-1,:]) * sqrtDt * W2[i-1,:])

R = np.sqrt(V)*sqrtDt*W1 - 0.5*V*dt
C = np.cumsum(R, axis=0)
S = np.exp(C)
A = np.mean(S, axis=0)
P = np.mean(np.maximum(A - K, 0))
```

In this model, the variance is a mean-reverting stochastic process. This slows down the simulation of the model.

Similarly the tensorflow implementation can be adjusted

```
def updateV(prevV, currW):
    return tf.abs(prevV + kappa * (theta - prevV) * dt +
        alpha * tf.sqrt(prevV) * sqrtDt * currW)

self.n = tf.placeholder(tf.int32)
W1t = tf.random_normal((nT, self.n))
W2t = rhobar*tf.random_normal((nT, self.n)) + rho*W1t

V0t = tf.fill([1, self.n], V0)
Vt = tf.scan(HestonTf.updateV, tf.concat([V0t, W2t[:-1,:]], 0))

Rt = tf.multiply(tf.sqrt(Vt), sqrtDt*W1t) - 0.5*Vt*dt
Ct = tf.cumsum(Rt, axis=0)
St = tf.exp(Ct)
At = tf.reduce_mean(St, axis=0)
self.Pt = tf.reduce_mean(tf.maximum(At - K, 0))
```

The scan function behaves like a while loop. It simulates the mean reverting variance process.

We compare the run times for pricing the Asian option also for this model using the same approach as before. Table 5 presents the results. The simulation in this model are slower than in the Black Scholes model because of the mean-reverting variance process. The slowdown is biggest for tensorflow gpu. But this implementation is still 30 times faster than the numpy implementation for 250000 simulations.

num sims	numpy	cpu	gpu
10000	0.49	0.29	0.11
25000	1.19	0.50	0.13
50000	2.37	0.90	0.16
100000	4.74	1.66	0.21
250000	12.2	4.17	0.41

Table 5: run times (in s) for pricing an Asian option in the Heston model via numpy, tensorflow cpu and gpu

Tensorboard is tensorflow's tool for visualization. In it, we can visualize the graph of a calculation or some statistics (eg convergence time series). The calculation graph is the topological ordering of operations. Figure 4 shows the graph of the Heston model where we introduced some scopes for the spot process and the payoff. The spot process contains another scope for the variance process. The figure shows some details of the spot process.

Next, we consider the Heston model with local volatility. The local volatility model is nonparametric and matches by construction the market data via a local volatility surface σ_{LV}. The Heston model can be extended to also match the market data using the local volatility surface. This model is described by

$$\begin{aligned} \frac{dS_t}{S_t} &= \sqrt{V_t}\frac{\sigma_{LV}(S_t,t)}{\sqrt{\mathbb{E}(V_t|S_t)}}dW_t^1 \\ dV_t &= \kappa(\theta - V_t)dt + \alpha\sqrt{V_t}dW_t^2 \end{aligned}$$

with Wiener processes W^1 and W^2 correlated by ρ.

As we focus only on the implementation, we simplify and use a constant local volatility surface $\sigma_{LV}(S_t,t) = \bar{\sigma}$. Additionally, we approximate the conditional expected value simply by a linear regression. This regression makes the model computationally more expensive.

The implementation in numpy uses a loop in order to linearly regress the last variances on the last spot prices.

```
W1 = np.random.randn(nT, n)
W2 = rhobar*np.random.randn(nT, n) + rho*W1

V = np.full((nT, n), V0)
```

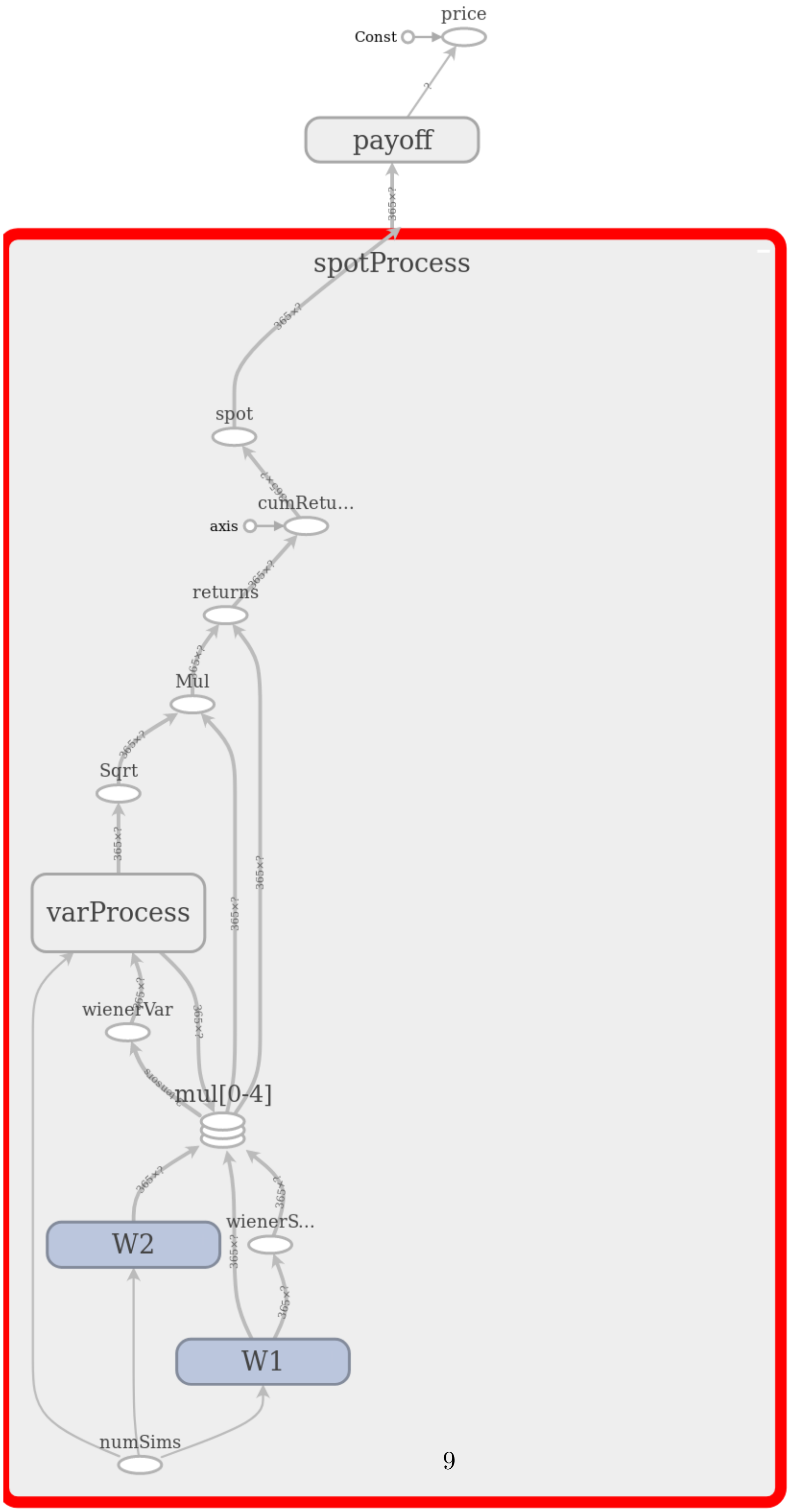


Figure 4: Graph of the Heston model

```
for i in range(1, nT):
    V[i,:] = np.fabs(V[i-1,:] + kappa * (theta - V[i-1,:])*dt +
        alpha * np.sqrt(V[i-1,:]) * sqrtDt * W2[i-1,:])

reg_coeff = np.array([V0, 0, 0]).T
X = np.ones((3, n))
logS = np.zeros([1, n])
S = np.zeros((nT, n))
for i in range(nT):
    var = V[i,:] * sigmabar*sigmabar / np.maximum(0.0001, np.matmul(X.T, reg_coeff))
    logS = logS + np.sqrt(var) * sqrtDt * W1[i,:] - 0.5*var*dt
    S[i,:] = np.exp(logS)
    X = np.vstack([np.ones(n), S[i,:], np.square(S[i,:])])
    reg_coeff = np.matmul(inv(np.matmul(X, X.T)), np.matmul(X, V[i,:]))

A = np.mean(S, axis=0)
P = np.mean(np.maximum(A - K, 0))
```

For the tensorflow implementation, we use a while loop and store the results in a tensor array.

```
W1t = tf.random_normal((nT, n))
W2t = rhobar*tf.random_normal((nT, n)) + rho*W1t

V0t = tf.fill([1, n], V0)
Vt = tf.scan(HestonLVTf.updateV, tf.concat([V0t, W2t[:-1,:]], 0))

def cond(i, _1, _2, _3, _4):
    return i < nT

def body(i, X, reg_coeff, lastLogS, ta):
    condExp = tf.reshape(tf.maximum(0.0001, tf.matmul(X, reg_coeff)), [n])
    var = tf.divide(sigmabar*sigmabar * Vt[i,:], condExp)
    logS = lastLogS + tf.multiply(tf.sqrt(var), sqrtDt * W1t[i,:]) - 0.5*var*dt
    ta = ta.write(i, logS)
    S = tf.exp(logS)
    X = tf.stack([tf.ones(S.shape), S, tf.square(S)], axis=1)
    reg_coeff = tf.matmul(tf.matrix_inverse(tf.matmul(X, X, transpose_a=True)),
        tf.matmul(X, tf.reshape(Vt[i,:],[n,1]), transpose_a=True))
    return i+1, X, reg_coeff, logS, ta

ta = tf.TensorArray(dtype=tf.float32, size=nT)
_0, _1, _2, _3, ta = tf.while_loop(cond, body,
    [0, tf.ones((n, 3)), tf.constant([[V0], [0], [0]]), tf.zeros(n), ta])

Ct = ta.stack()
```

```
St = tf.exp(Ct)
At = tf.reduce_mean(St, axis=0)
self.Pt = tf.reduce_mean(tf.maximum(At - K, 0))
```

Table 6 presents the comparison of run times for pricing the Asian option in Heston model with local volatility. The simulations in this model are slow because of the regression at each time step. Tensorflow is only 7 times faster than the numpy implementation for 250000 simulations.

num sims	numpy	cpu	gpu
10000	0.66	0.71	0.70
25000	1.57	1.29	0.47
50000	3.18	2.18	0.68
100000	6.40	3.77	1.07
250000	16.7	8.36	2.37

Table 6: run times (in s) for pricing an Asian option in the Heston model with local vol via numpy, tensorflow cpu and gpu

The implementation of this model was kept simple for illustration. The implementation can be changed by regressing not on every time step or by precomputing the conditional expected value (by regressing also on the five model parameters). The local volatility function also plays a role. Another approach is to fit a parametric function $\sigma_p(S_t, t)$ such that the overall volatility in the model is $\sqrt{V_t}\sigma_p(S_t, t)$ and market prices are matched.

4 Autodiff and distributed computing

Sometimes, you need derivatives of the option price, eg for model calibration or for hedge sensitivities. As the derivatives cannot be computed manually, numerical methods are often used. They require $p+1$ evaluations for a one-sided finite difference for p parameters and they give only approximative derivatives which depend a lot on the bump size.

Tensorflow implemented reverse-mode autodifferentiation. This requires one forward and one backward evaluation of the graph for any number of input parameters and returns exact derivatives for all parameters. Hence it is fast if there are many parameters. Tensorflow also allows to make your own operations compatible with autodiff. The implementation of reverse-mode autodiff computes the partial derivatives with regards to each node in reverse order. It is based on the chain-rule.

We look at implementations of derivatives in the Heston model focusing on the parameters V_0 and α as illustration. In numpy, we use finite differences. In tensorflow, we modify the code for pricing in order to use autodiff. To this end, we make V_0 and α to tensorflow variables. Such variables require a node for initialization which has to be run first. The derivatives are defined via a gradients node:

```
alphav = tf.Variable(alpha)
V0v = tf.Variable(V0)
...
Pt = tf.reduce_mean(tf.maximum(At - K, 0))
Deriv = tf.gradients(Pt, [V0v, alphav])
init = tf.global_variables_initializer()
...
init.run()
d = sess.run(Deriv)
```

Table 7 compares the run times in numpy, tensorflow cpu and tensorflow gpu for the computation of partial derivatives. We observe that the run times for tensorflow are the same independently of the number of variables. This leads to a speedup of factor 50 for tensorflow gpu versus numpy for the joint calculation. Moreover, the derivatives with reverse-mode autodiff are exact while finite difference gives approximations.

gradients	numpy	cpu	gpu
$dP/dV0$	9.52	4.02	0.30
$dP/dV0, dP/d\alpha$	14.5	4.09	0.31

Table 7: run times (in s) for computing gradients for an Asian option in the Heston model using 100000 simulations

Forward-mode autodiff is another approach for autodifferentiation, it is not implemented in tensorflow but it might be used via a trick. Reverse-mode autodiff is more efficient than forward-mode autodiff for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ if $m << n$. In neural nets, a main application for tensorflow, the inputs are often higher-dimensional than the outputs. Table 8 illustrates higher dimensional outputs for reverse-mode autodiff where the sum of derivatives could stand for a mse.

gradients	numpy	cpu	gpu
$dP_1/dV0, dP_2/dV0$	9.52	8.04	0.60
$dP_1/dV0 + dP_2/dV0$	9.52	4.09	0.31

Table 8: run times (in s) for computing gradients for Asian options P_1 and P_2 in the Heston model using 100000 simulations

In model calibration and other problems in finance, we often use optimization routines. Tensorflow offers several modern optimizers that converge in general faster than plain gradient descent.

If we want to mimize $f(x)$ then gradient descent makes updates $x_{n+1} = x_n - \eta \nabla f(x_n)$ which can be slow in flat regions. Momentum optimization updates instead a momentum $m_{n+1} = \beta m_n + \eta \nabla f(x_n)$ which is used for updating the parameter $x_{n+1} = x_n - m_{n+1}$. Here $\beta in(0,1)$ is a hyperparameter that controls the maximum speed up over plain gradient descent. Tensorflow offers this momentum optimizer and also an improved variant by Nesterov.

RMSProp is another optimizer in tensorflow. It scales down the gradient along the steepest dimensions based on the most recent iterations $s_{n+1} = \beta s + (1-\beta)\nabla f(x_n) \otimes \nabla f(x_n)$ ($\otimes$ represents element-wise multiplication) and $x_{n+1} = x_n - \eta \nabla f(x_n) \oslash \sqrt{s_{n+1} + \epsilon}$. This optimizer has been combined with the momentum optimizer into the Adam optimizer, adaptive moment estimation. All these optmizers can be used easily in tensorflow.

An application for autodiff and optimization might be the calibration of a variant of the Heston model with local volatility. Instead of computing

$$\frac{\sigma_{LV}(S_t, t)}{\sqrt{\mathbb{E}(V_t|S_t)}}$$

we approximate this surface by an effective local volatility surface $\sigma_p(S_t, t)$. This can be calculated by minimizing the sum of squared errors to market prices. This problem has many parameters for the parametric effective local volatility and only one output parameter. Hence reverse-mode autodiff speeds up the calculation as discussed above.

In order to speed up computations of option prices, it is common pratice to distribute the calculations across several machines/cores. Tensorflow makes it easy to distribute computations across several devices on one machine or across several machines. We have shown above how to pin a graph to a device. If we dont specify a device, then tensorflow will use some simple placement rules. Many operations have implementations on both cpu and gpu but not all, eg the singular value decomposition seems to have no gpu implementation or kernel. If operations are placed on the same device, they may run in parallel in different threads if there are no dependencies. You can also use multiple devices across multiple servers. Again operations can be pinned easily across tasks. Tensorflow allows to share variables across servers and sessions. It also provides tools for asynchronous communication between sessions.

5 Summary

We showed how to implement some option pricing models on gpu. We used tensorflow in python which is based on C++. But tensorflow can also be used directly in C++. The implementation in tensorflow is shorter and easier than in cuda. Additionally, tensorflow has helpful tools like tensorboard and timeline.

The speed gains compared to numpy depend on the problem, ie the model and the product. Our tests showed speed ups between 10 and 50 for pricing and 30 to 50 for gradients.

Tensorflow offers additionally automatic differentiation, modern optimization algorithms and easy handling of distributed computing.

We considered examples for illustrating option pricing. A production system for option pricing is more involved because there are additional layers, eg for market data and payoff evaluation. Moreover, we have looked only at a payoff based on one underlying. Hence the monte carlo is simplified. Additionally, we ignored some aspects, eg quality of random numbers or alternatives like quasi random numbers.

Overall, tensorflow seems to give significant speed ups over cpus while keeping the code and its coding manageable.